Module 7: Object-Oriented Detailed Design

Software Engineering

Faculty of Computing,
Universiti Teknologi Malaysia

Objectives

- To understand the relationship among analysis, design and implementation
- To comprehend object-oriented design principles
- To further understand object-oriented design using UML
- To elaborate the related diagrams from analysis in design phase in use case realization

References: Sommerville (2016), Arlow and Neustadt (2002), Satzinger (2011)

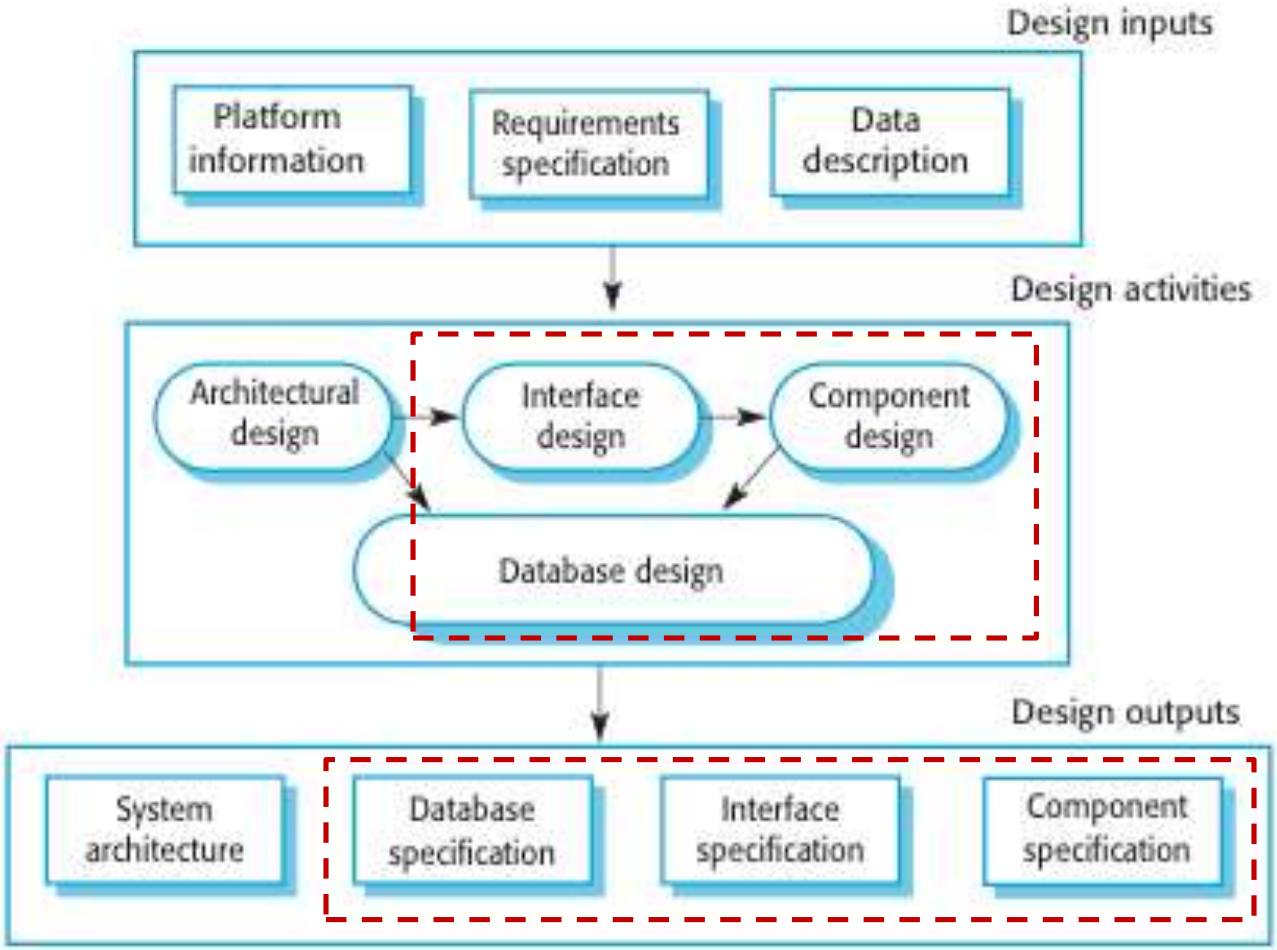
Objective 1:

RELATIONSHIPS AMONG ANALYSIS, DESIGN AND IMPLEMENTATION

Design and Implementation

- Software design and implementation is the stage in the software engineering process at which an **executable software system** is developed
- Software design and implementation activities are invariably **inter-leaved:**
  - Software **design is a creative activity** that identifies software components and their relationships, based on a customer's requirements
  - Implementation is the process of **realizing the design** as a program

Recap on Design Stage



Object-Oriented Design Process

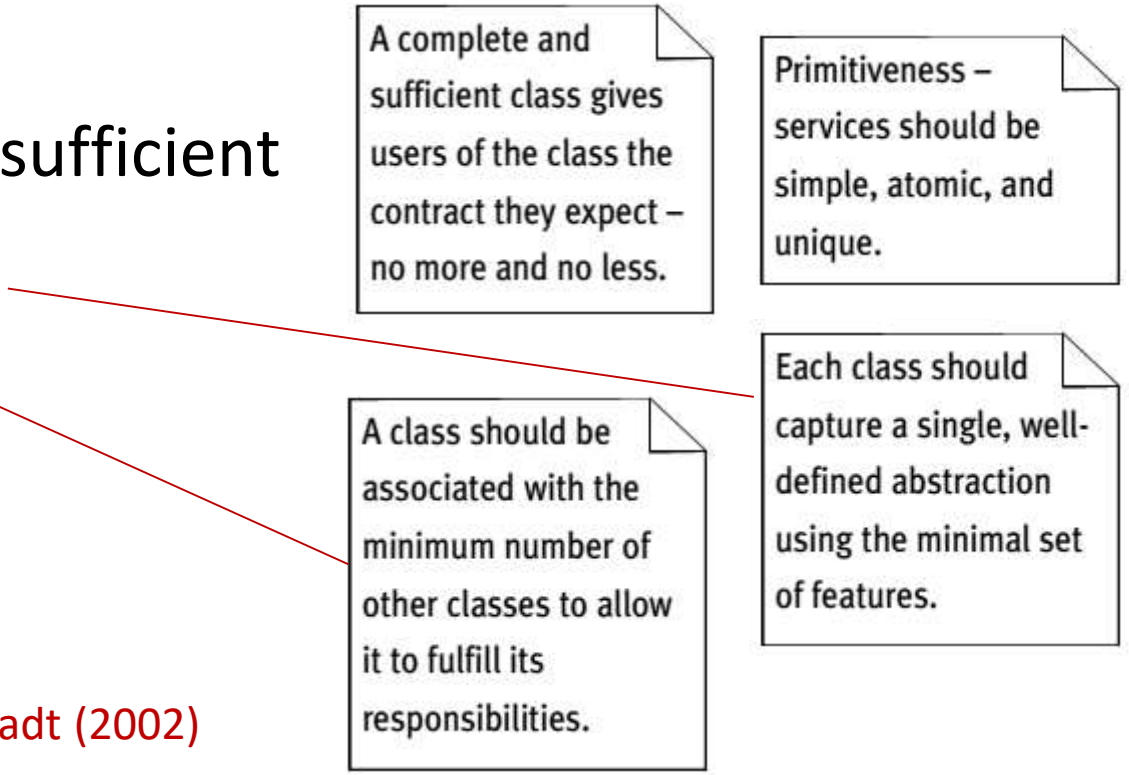
- Structured object-oriented design processes involve developing a number of **different system models**
- They **require a lot of effort** for development and maintenance of these models and for **small systems**, this **may not be cost-effective**
- However, for **large systems** developed by different groups, design models are an important **communication mechanism**

Objective 2:

OBJECT-ORIENTED DESIGN PRINCIPLES

Well-Formed Design Class

- Four minimal characteristics that a design class must have to be considered **well-formed**:
  - complete and sufficient
  - primitive
  - high cohesion
  - low coupling



Source: Arlow and Neustadt (2002)

Some Fundamental Design Principles…

- Encapsulation
  - Each object is a self-contained unit containing both data and program logic
- Object reuse
  - Standard objects can be used over and over again within a system
- Information hiding
  - Data associated with an object is not visible
  - Methods provide access to data

Some Fundamental Design Principles

- Navigation visibility
  - Describes which objects can interact with each other
- Coupling
  - Measures how closely classes are linked
- Cohesion
  - Measures the consistency of functions in a class
- Separation of responsibilities
  - Divides a class into several highly cohesive classes

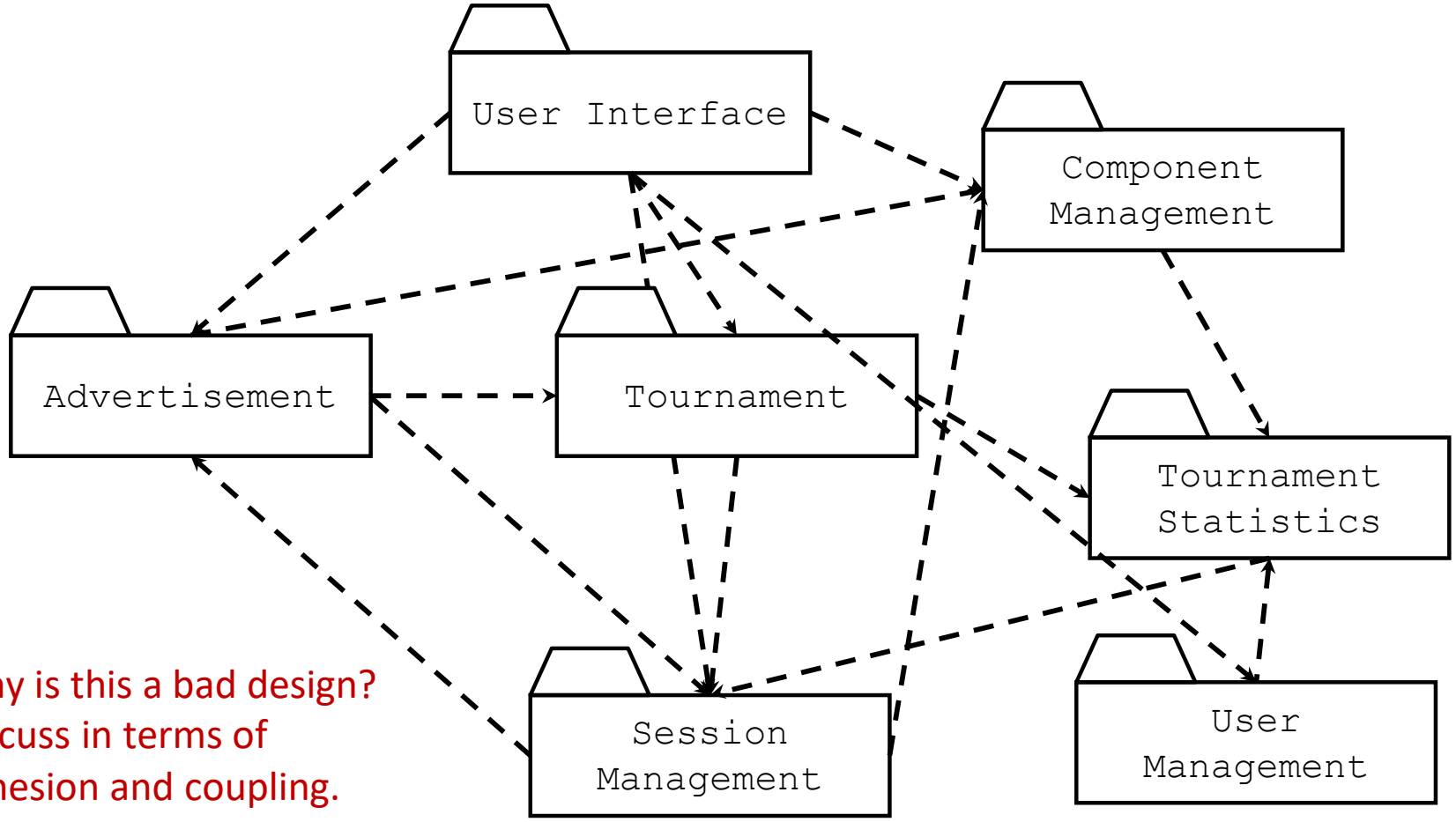
Coupling and Cohesion of Subsystems

Good Design

- Goal: Reduce system complexity while allowing change
- Coherence measures dependency among classes
  - **High cohesion:** The classes in the subsystem perform similar tasks and are related to each other via associations
  - Low cohesion: Lots of miscellaneous and auxiliary classes, no associations
- Coupling measures dependency among subsystems
  - High coupling: Changes to one subsystem will have high impact on the other subsystem
  - **Low coupling:** A change in one subsystem does not affect any other subsystem

Source: Bennett et al. (2010)

Example of a Bad Subsystem Decomposition



Why is this a bad design? Discuss in terms of cohesion and coupling.

Source: Bennett et al. (2010)

How to Achieve High Cohesion

- High cohesion can be achieved if **most of the interaction is within subsystems**, rather than across subsystem boundaries
- Questions to ask:
  - Does one subsystem always call another one for a specific service?
    - Yes: Consider moving them together into the same subsystem
  - Which of the subsystems call each other for services?
    - Can this be avoided by restructuring the subsystems or changing the subsystem interface?

How to Achieve Low Coupling

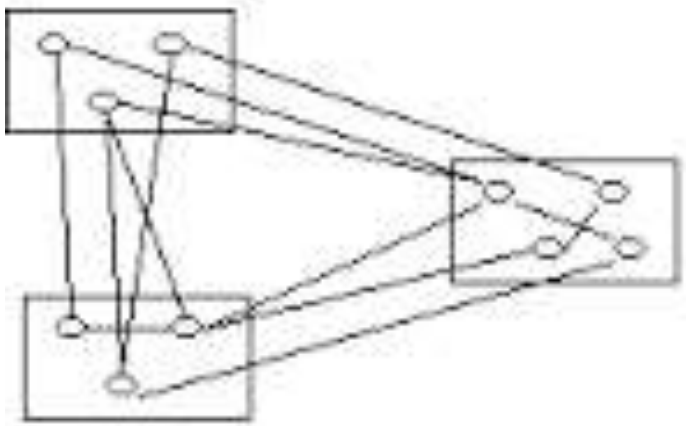
- Low coupling can be achieved if a **calling class or interface does not need to know anything about the internals** of the called class (Principle of information hiding, Parnas)
- Questions to ask:
  - Does the calling class really have to know any attributes of classes in the lower layers?
  - Is it possible that the calling class calls only operations of the lower level classes?

David Parnas, *1941,
Developed the concept of
modularity in design

Comparison

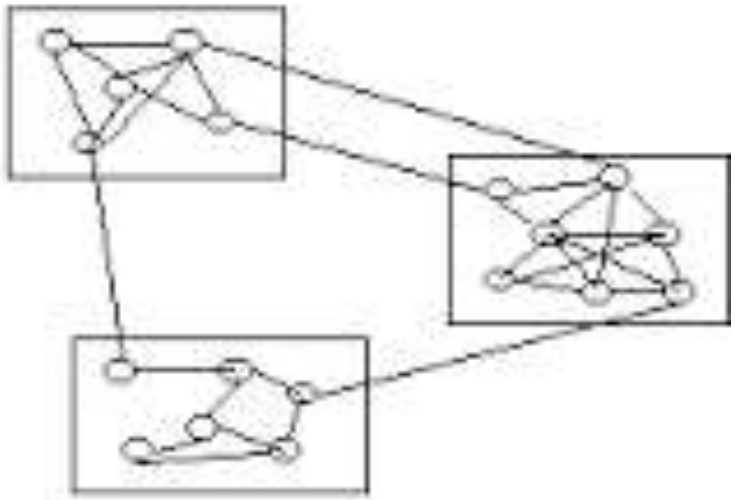
Bad

Low Cohesion
High Coupling



Good

High Cohesion
Low Coupling



Objective 3:

OBJECT-ORIENTED DESIGN MODELS USING UML

Design Models Artefacts

- Design model involves **an elaboration of the analysis model** where we add detail and specific technical solutions
- The design model contains the same sorts of things as the analysis model, but **all the artefacts are more fully formed** and must now include **implementation details**
- E.g. An analysis class may be little more than a sketch with few attributes and only key operations (or just a domain model)
- A design class, however, must be fully specified – all attributes and operations (including return types and parameter lists) must be complete

Design Models: UML Diagrams

- Design subsystems Package Diagram
- Design classes Class Diagram also known as Design Class Diagram
- Interfaces Component Diagram (as discussed in Module 6: Architectural Design)
- Use case realizations Sequence Diagram also known as Design Sequence Diagram (and its relation to Design Class Diagram)
- Deployment diagram (implementation)

Design Models (our focus)

- Design subsystems Package Diagram
- Design classes Class Diagram
- Interfaces (already discussed in Module 6)
- Use case realizations Sequence Diagram
- Deployment diagram (implementation)

Object-Oriented Design Models

- Identify all **objects that must work together** to carry out a use case
- Package diagrams denote which **classes work together as a subsystem**
- Divide objects into groups for a **multilayer design**
- Sequence diagrams **describe the messages** that are sent between objects
- Design class diagrams document and describe the programming classes (implementation)
- Design information is primarily derived from domain model and sequence diagram from the analysis stage

Design subsystems Package Diagram

Design classes Class Diagram

Use case realizations Sequence Diagram

Deployment diagram (implementation)

DESIGN SUBSYSTEMS: PACKAGE DIAGRAM

Package Diagram: Structuring the Major Components

- Associate **classes of related groups** or **related use case**
- One option is to separate the **view, domain, and data access layers** into separate packages
- Indicate **dependency relationships**:
  - Shows which elements affect other elements in a system
  - May exist between packages, or between classes within packages
- Packages can be nested

Multilayer Design

- Focus on a basic multilayer design that consists of the view layer **realization of use cases specification** of all detailed system processing for:
  - User interface classes
  - Domain layer (problem domain classes from the domain model class diagram), controller classes
  - Data layer (database access classes)

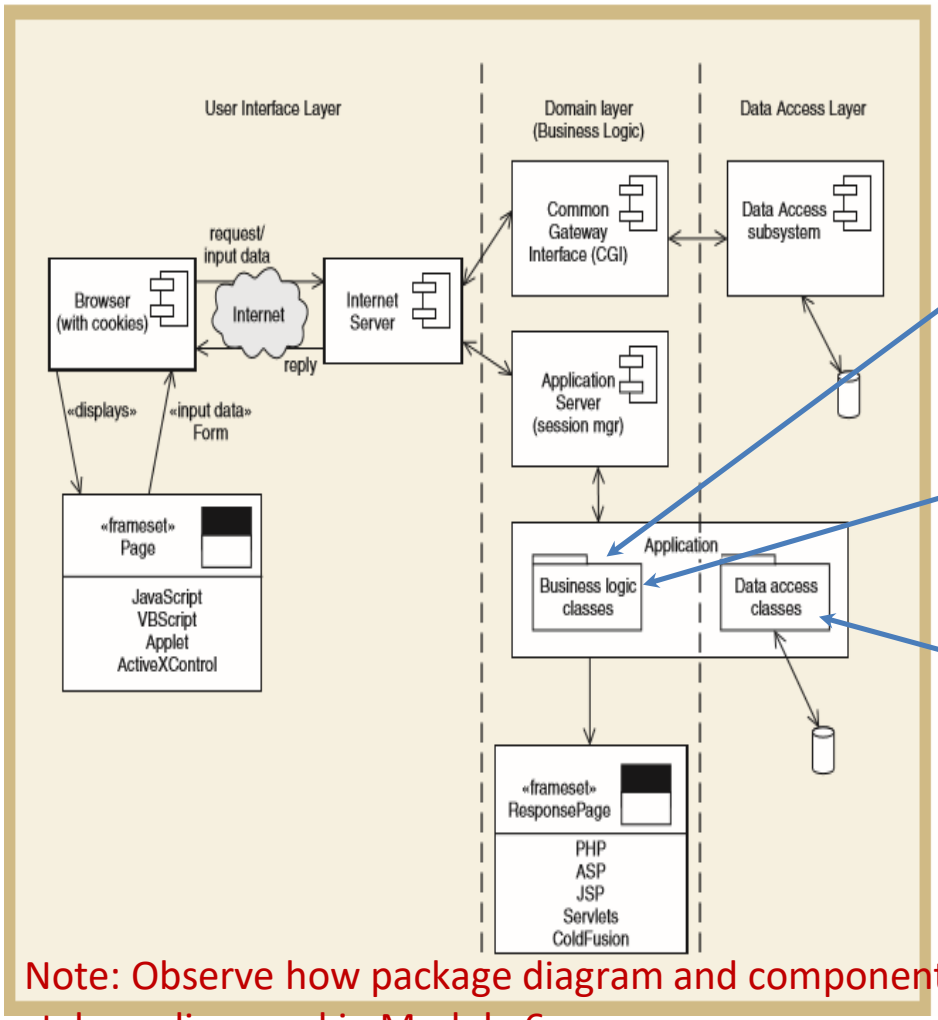
Heuristics to Identify Subsystem

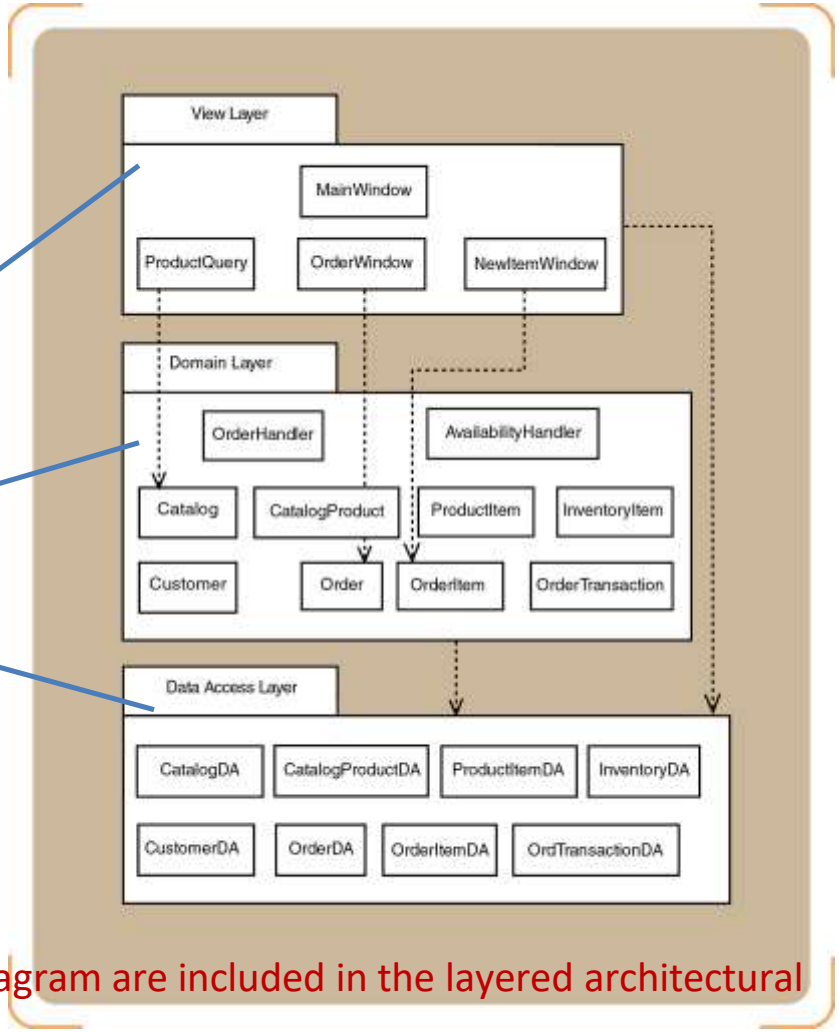
- Assign **objects identified in one use case** into the same subsystem
- Minimize the number of associations crossing subsystem boundaries
- All objects in the **same subsystem** must be **functionally related**

Source: Bennett et al. (2010)

Example of Design Subsystems:

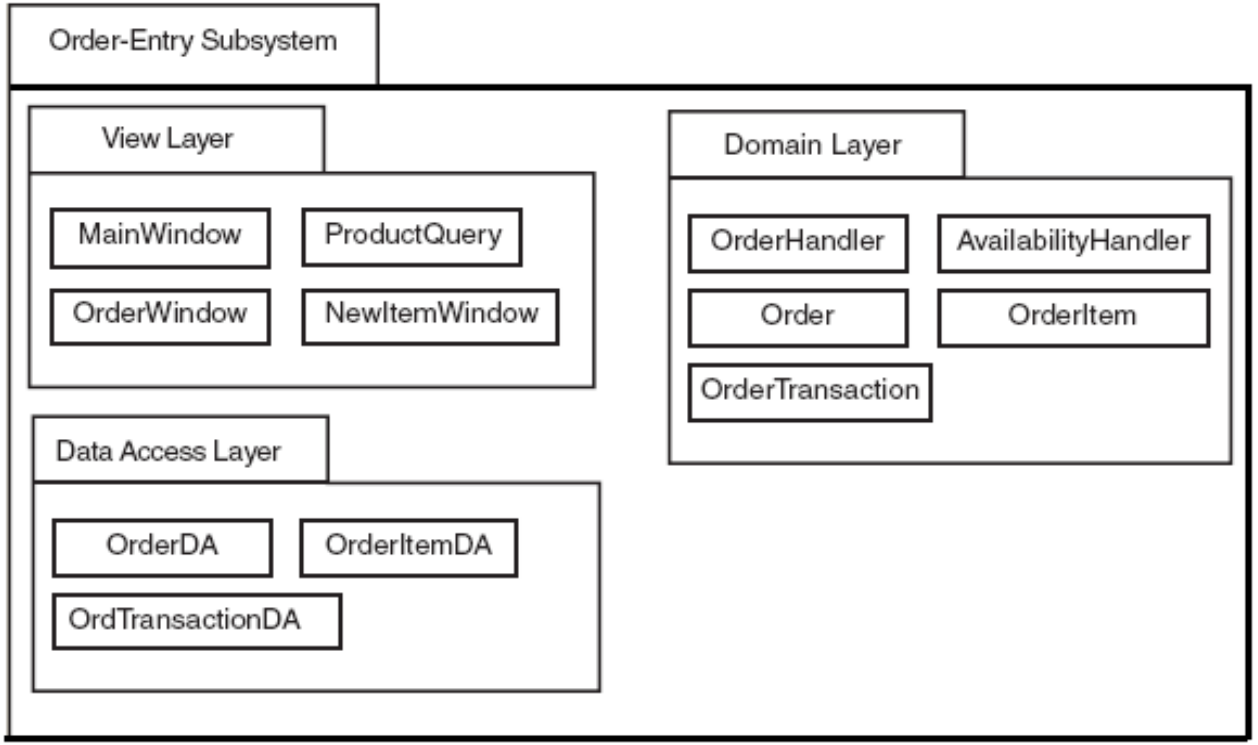
Combination of Layered (user interface, domain layer and data access layer), Subystems and Components





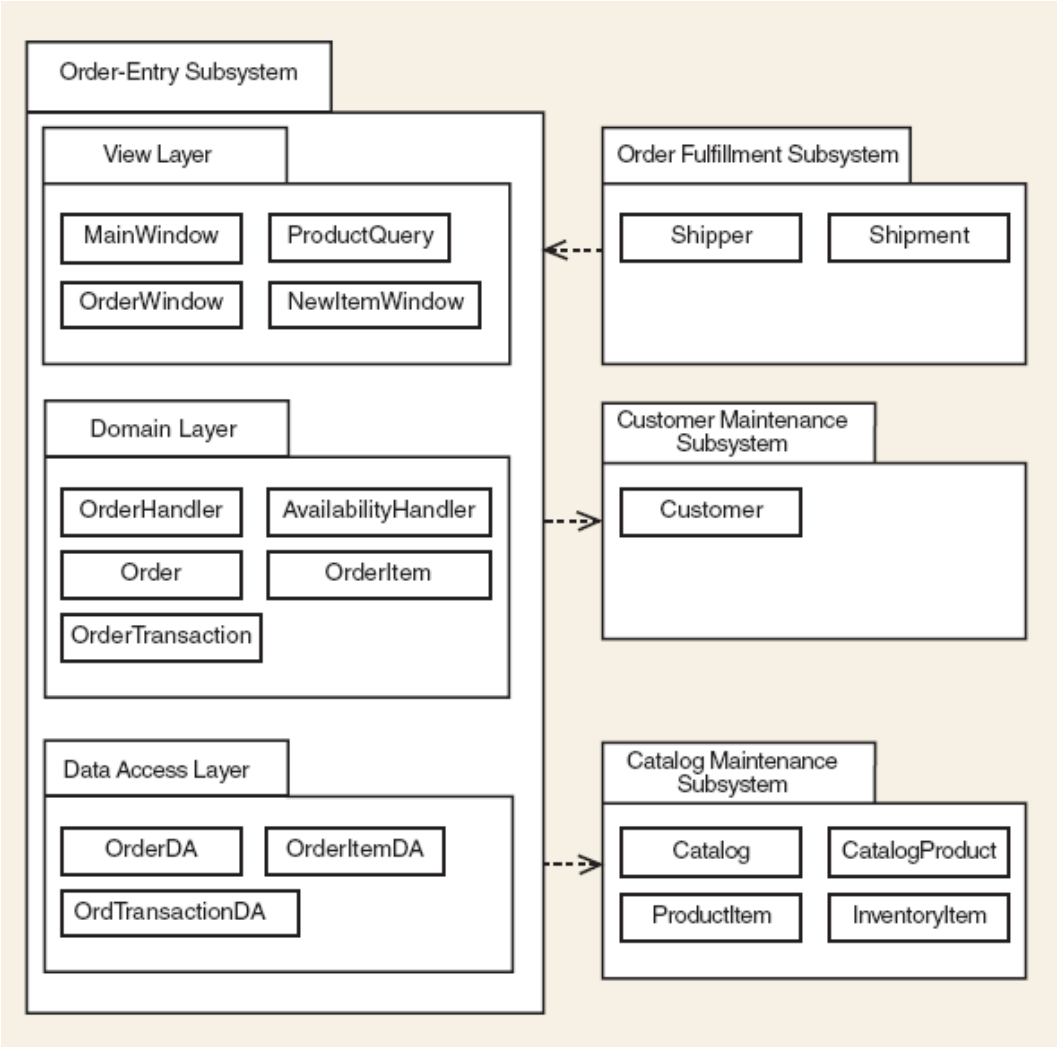
Note: Observe how package diagram and component diagram are included in the layered architectural style as discussed in Module 6

Example: Three-Layer Package Diagram for Order-Entry Subsystem



Note: Domain Layer also includes the controllers (OrderHandler and AvailabilityHandler)

Example: Order Entry Subsystem and the interaction with other subsystems



Design subsystems Package Diagram

Design classes Class Diagram

Use case realizations Sequence Diagram

Deployment diagram (implementation)

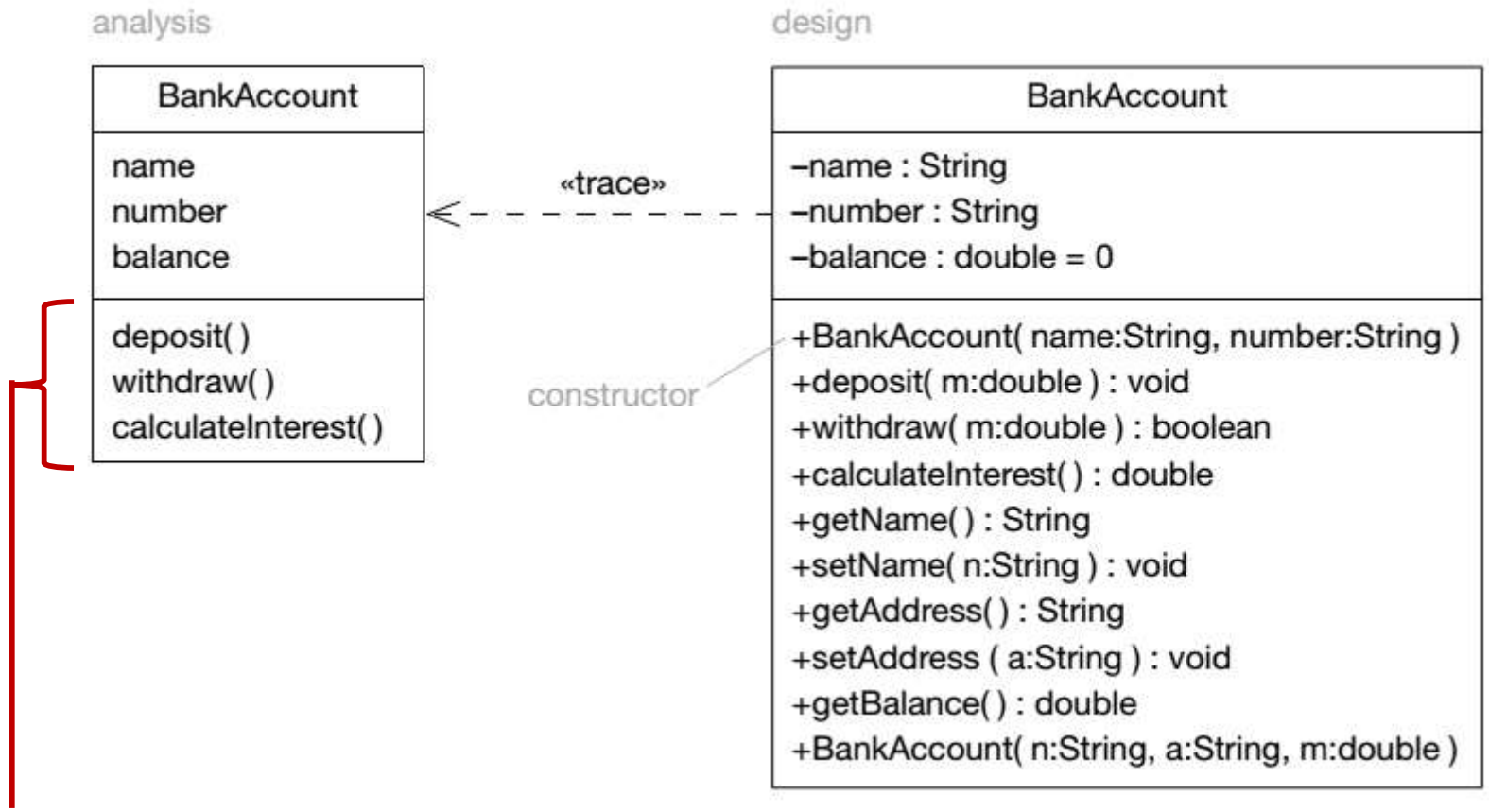
DESIGN CLASSES: CLASS DIAGRAM

Anatomy of a Design Class

- In **analysis**, we just capture the required behaviour of the system without worrying at all about how this behaviour is going to be implemented
- In **design**, we have to specify exactly how the class will fulfil its responsibilities by doing the following:
  - Complete the set of attributes and fully specify them including name, type, visibility and (optionally) a default value
  - Turn the operations specified in the analysis class into a complete set of one or more methods

Note: In our SE scope, we only produce domain model in analysis without operations

Analysis vs. Design Class



Note: In our SE scope, we only produce domain model in analysis without operations

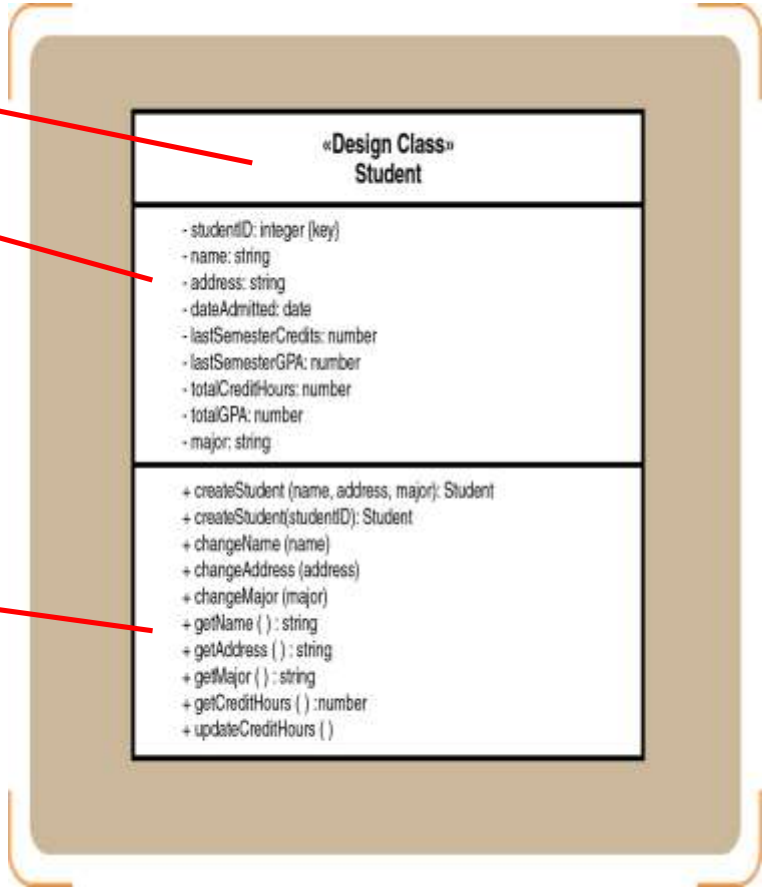
Design vs. Implementation

```
public class Student
{
      //attributes
      private int studentID;
      private String firstName;
      private String lastName;
      private String street;
      private String city;
      private String state;
      private String zipcode;
      private Date dateAdmitted;
      private float numberCredits;
      private String lastActiveSemester;
      private float lastActiveSemesterGPA;
      private float gradePointAverage;
      private String major;

      //constructors
      public Student (String inFirstName, String inLastName, String inStreet,
            String inCity, String inState, String inZip, Date inDate)
      {
            firstName = inFirstName;
            lastName = inLastName;
            ...
      }
      public Student (int inStudentID)
      {
            //read database to get values
      }

      //get and set methods
      public String getFullName ( )
      {
            return firstName + " " + lastName;
      }
      public void setFirstName (String inFirstName)
      {
            firstName = inFirstName;
      }
      public float getGPA ( )
      {
            return gradePointAverage;
      }
      //and so on

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

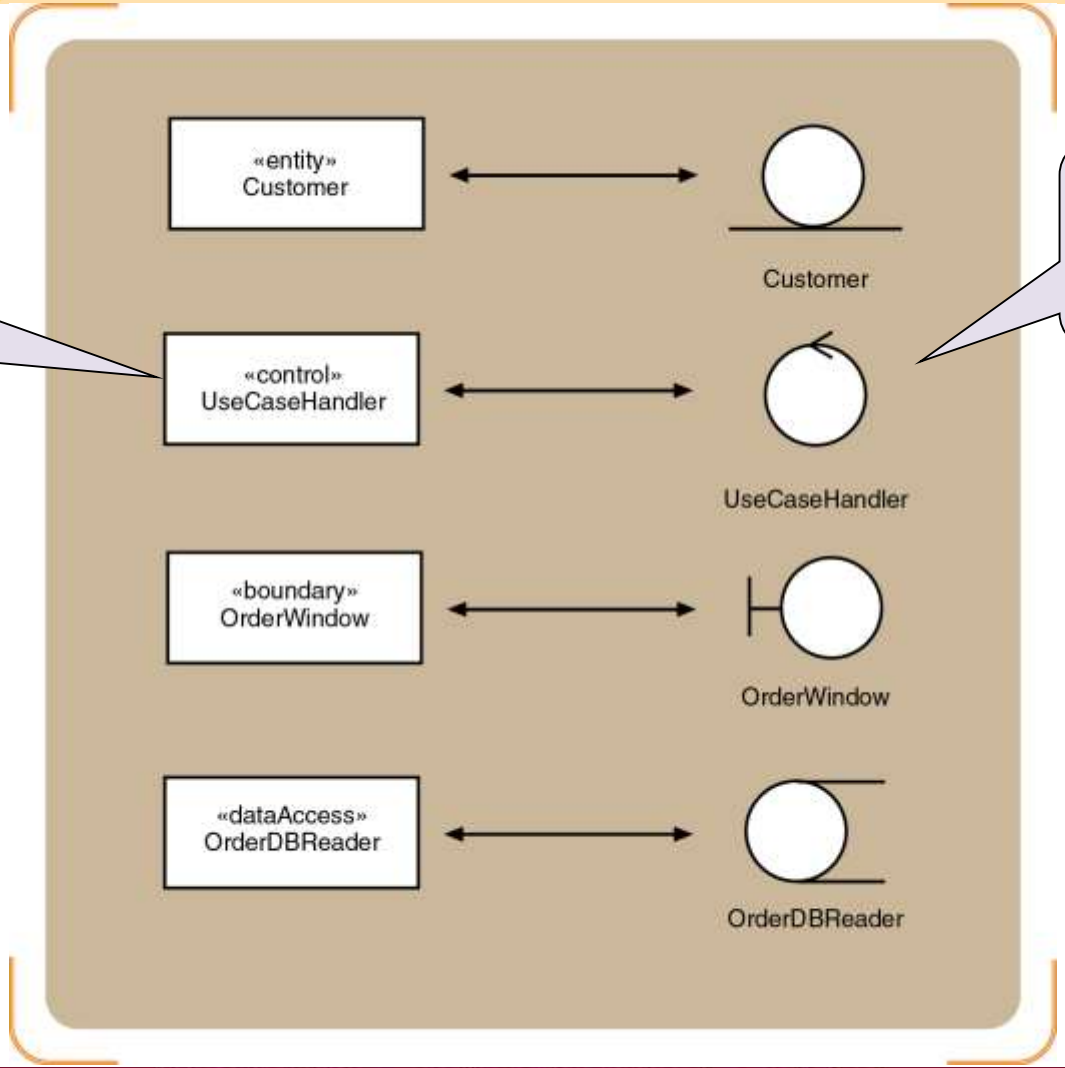


Design Class Symbols Using Stereotypes

- Stereotypes
  - UML notation to categorize a model element as a certain type
- Two types of notation
  - Full notation with guillemets («»)
  - Shorthand notation with circular icons
- Standard stereotypes
  - Entity, control, boundary, data access

Note: This concept has been introduced in the analysis stage

Standard Stereotypes in Design Models



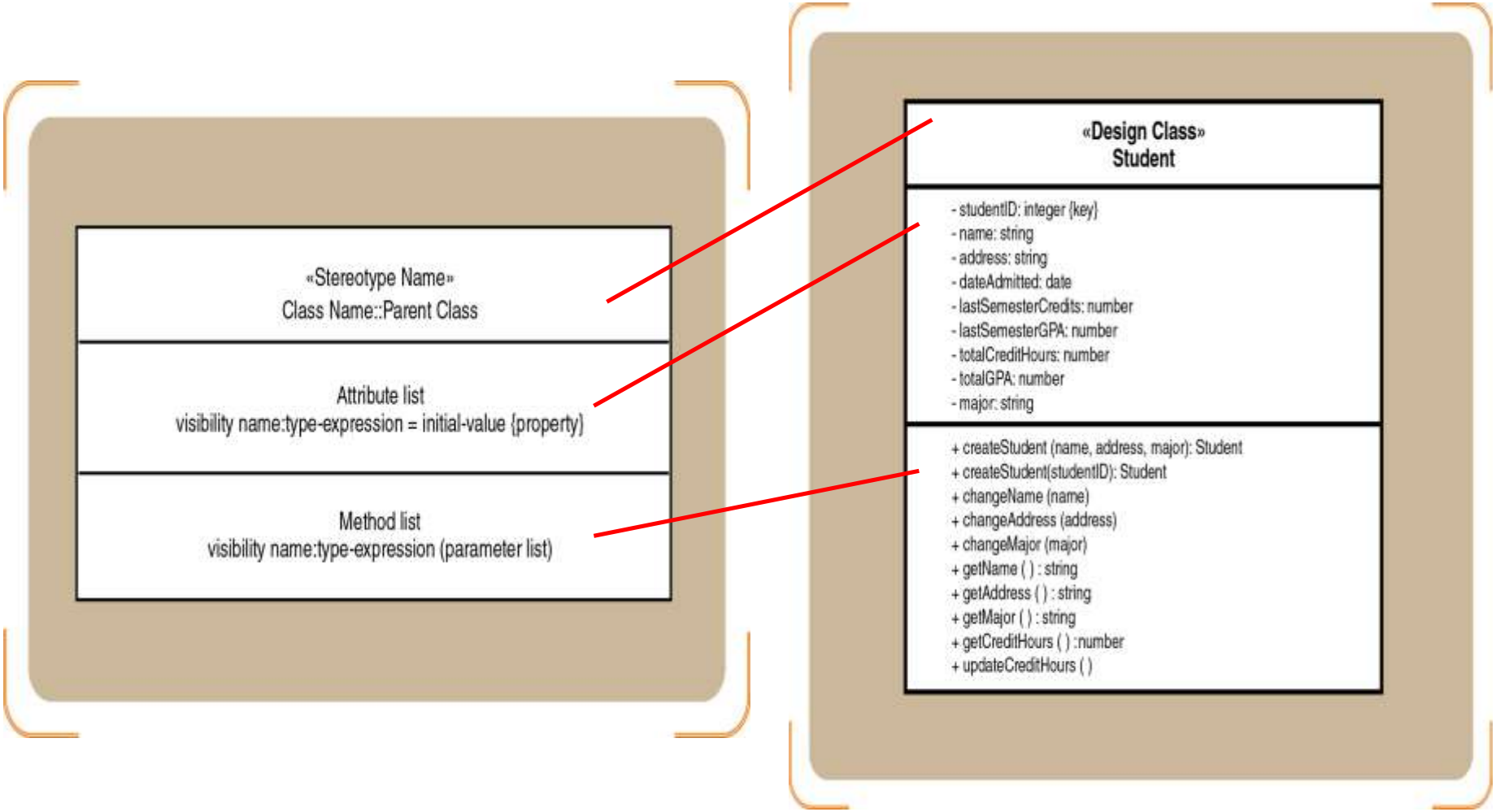
Full notation

Shorthand notation

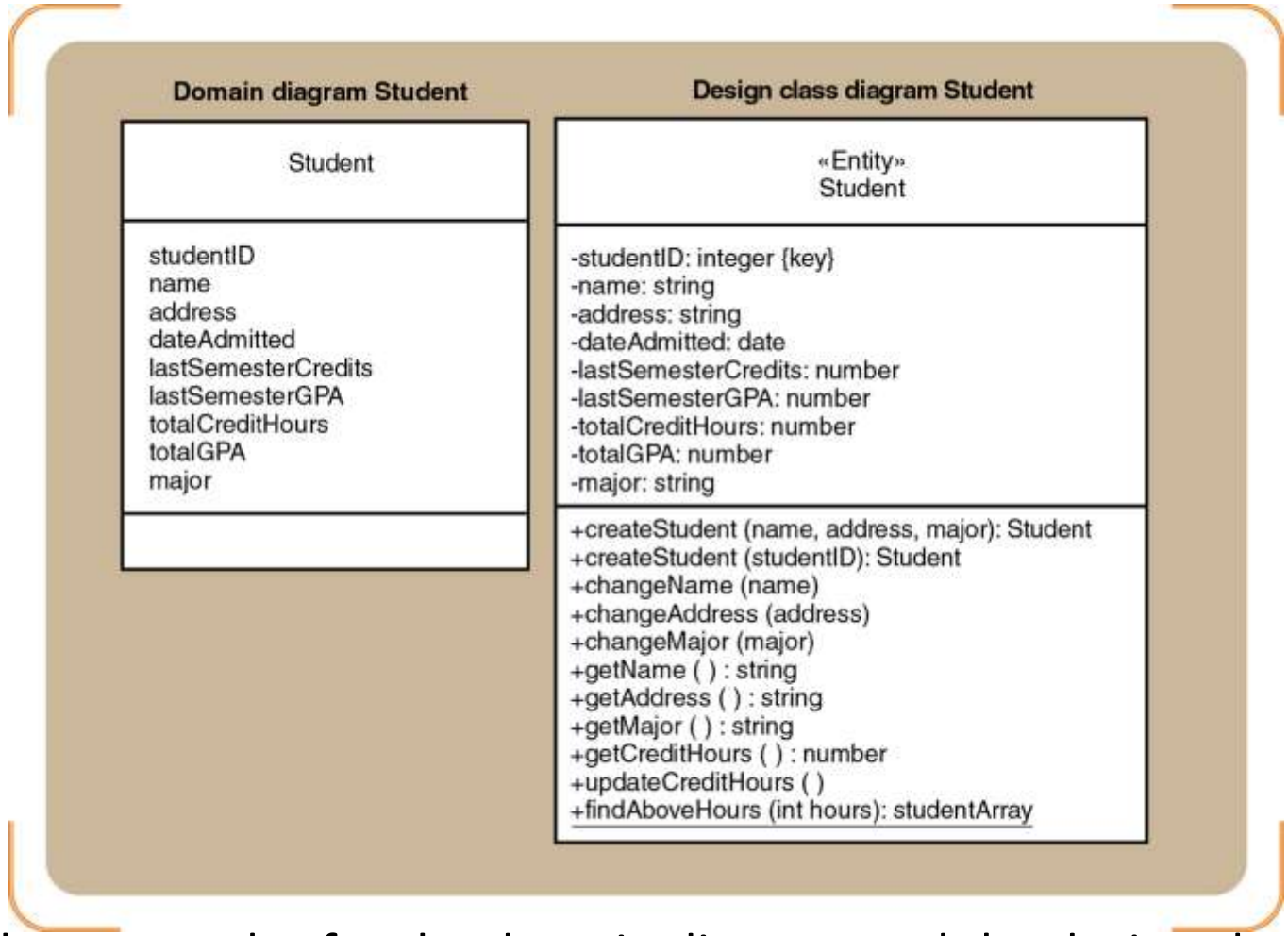
Design Class Notation

- Class name and stereotype information
- Attribute information
  - Visibility, type-expression, name, initial value, and properties
- Method signature
  - Visibility, name, type-expression, and parameter list
  - Use the entire signature to identify a method to distinguish between overloaded methods

Internal Symbols and Example



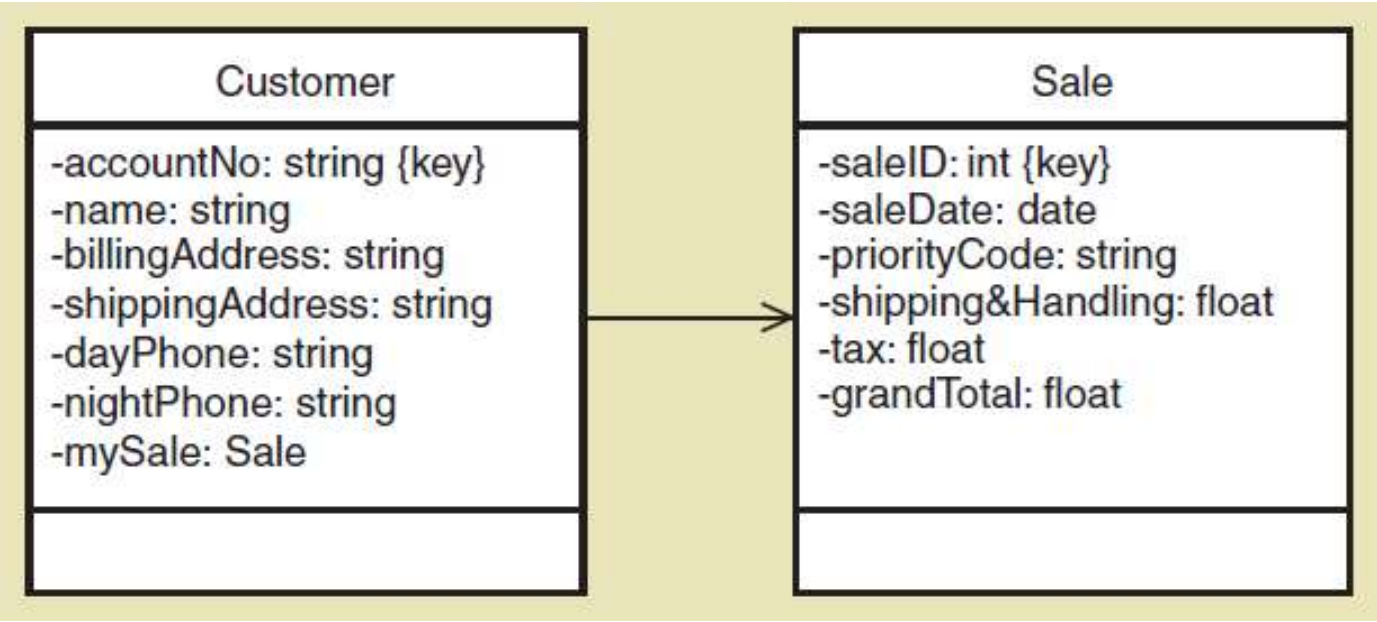
Domain Model vs. Design Class Diagram



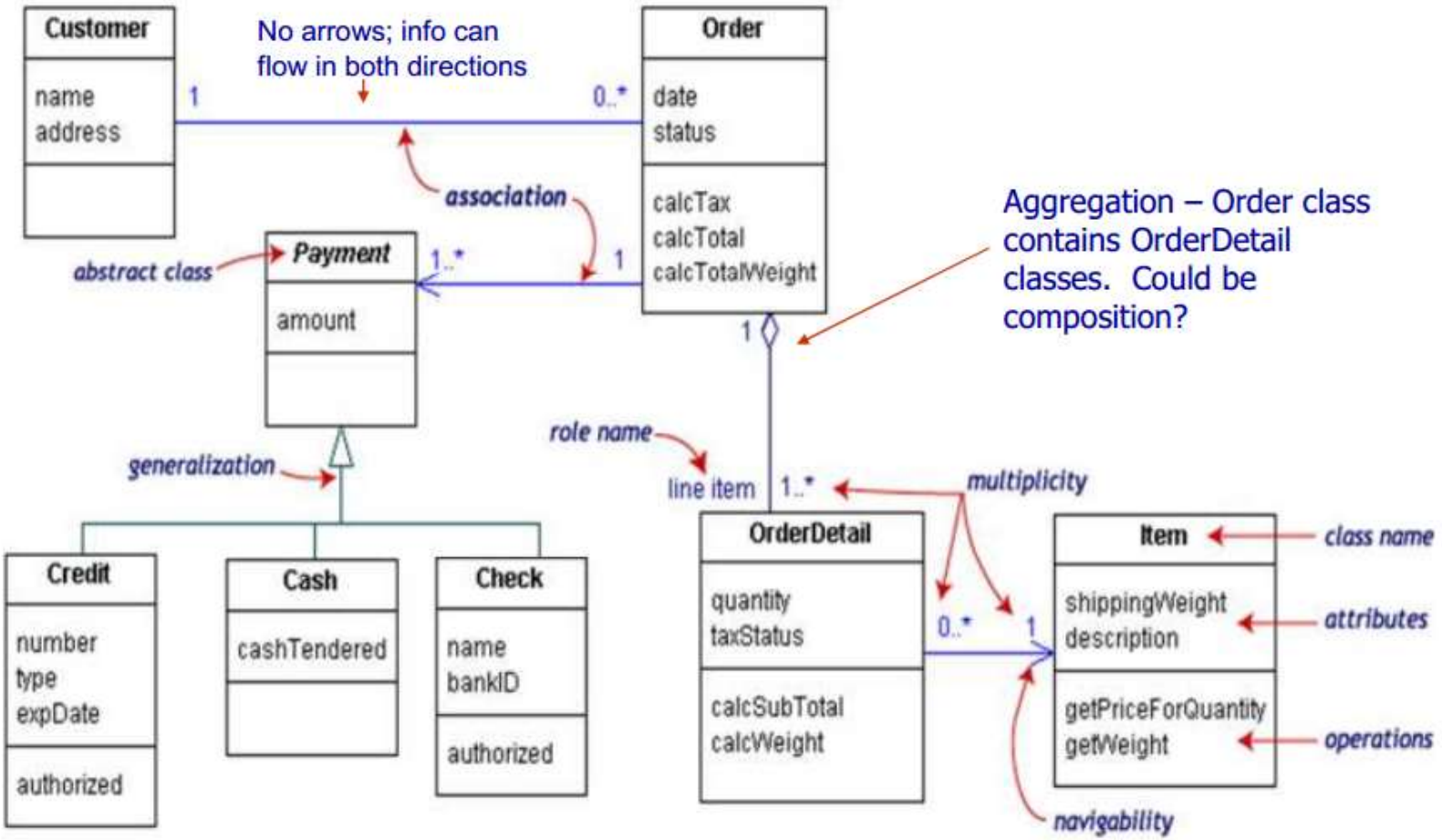
Student class examples for the domain diagram and the design class diagram

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 arrowhead on the association line—customer can find and interact with sale because it has `mySale` reference variable



Example of Class Diagram with Navigability



Source: https://softwareengineering.stackexchange.com/questions/195614/uml-class-diagram-confusion?noredirect=1&lq=1

Design subsystems → Package Diagram

Design classes → Class Diagram

Use case realizations → Sequence Diagram

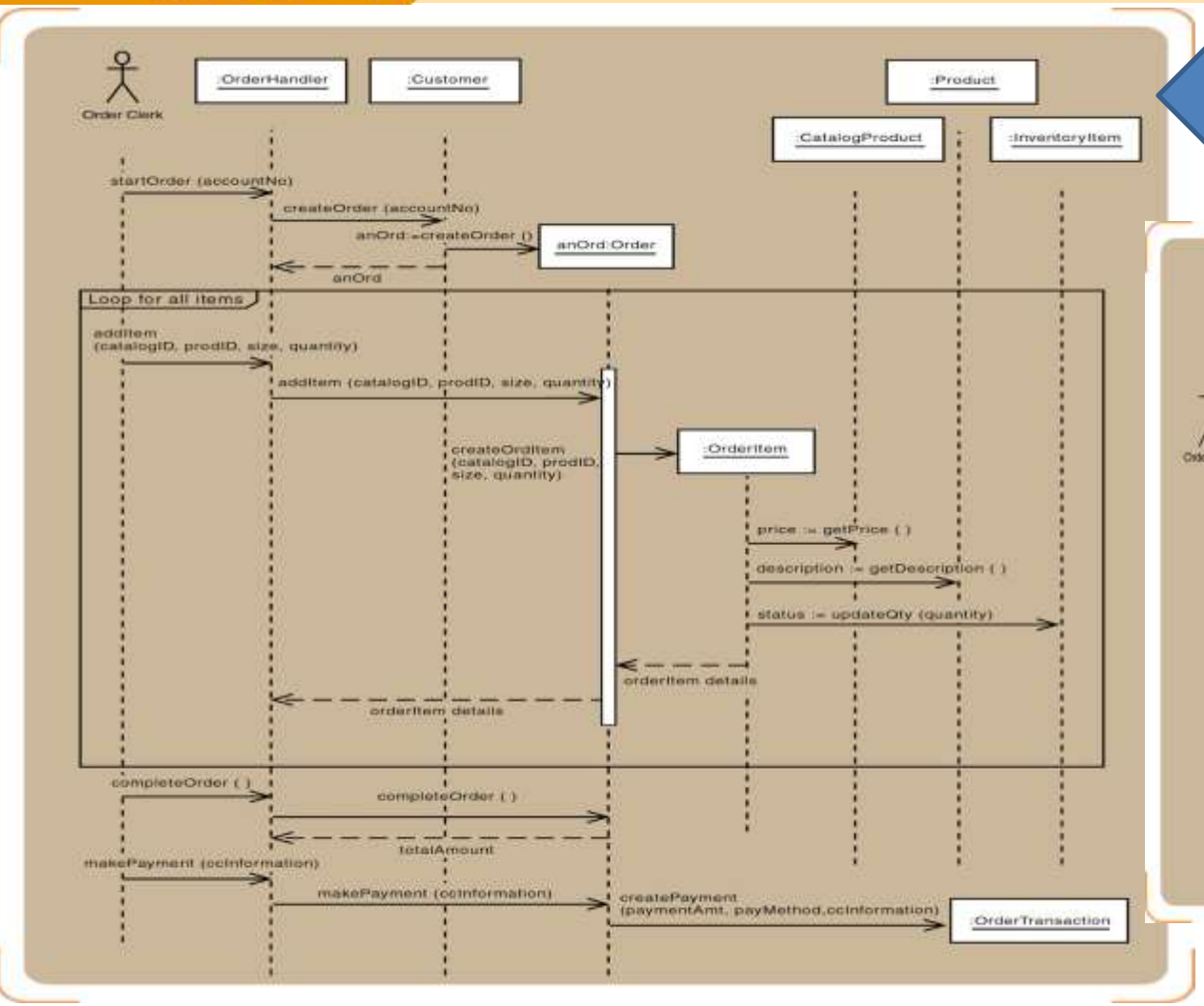
Deployment diagram (implementation)

USE CASE REALIZATIONS: SEQUENCE DIAGRAM

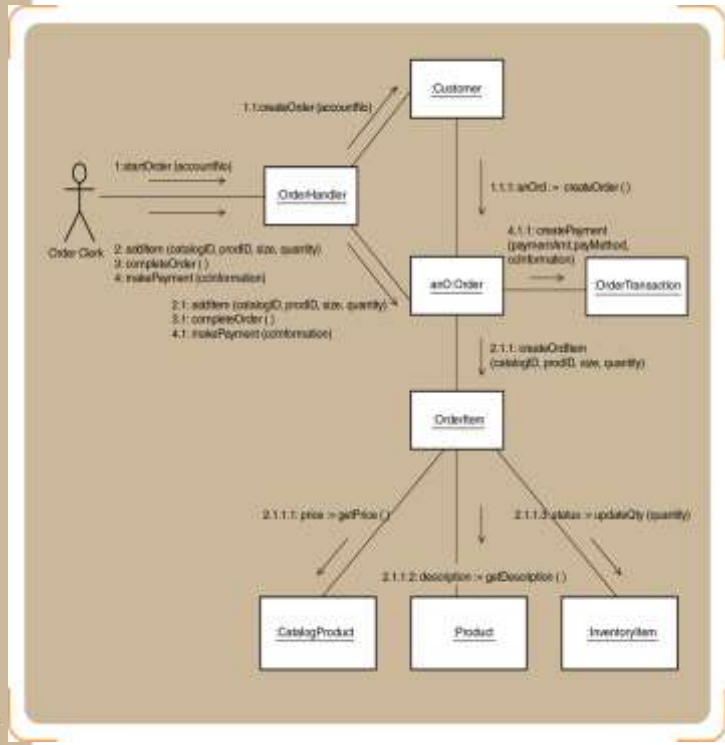
Interaction Diagrams: Realizing Use Cases and Defining Methods

- Interaction diagrams are at the heart of object-oriented design
- Realization of a use case:
  - Determine **what objects collaborate** by sending messages to each other
- Two types:
  - Sequence (Our focus)
  - Communication

Interaction Diagrams: Sequence and Collaboration (will be elaborated in use case realization)



Our focus



Design subsystems Package Diagram

Design classes Class Diagram

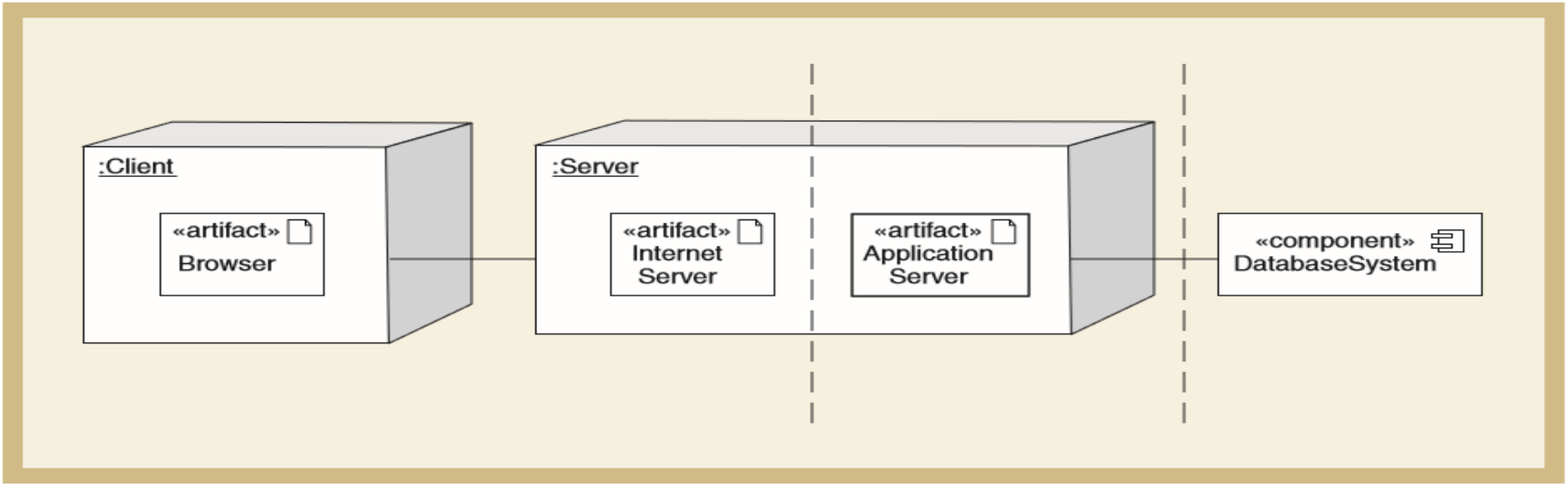
Use case realizations Sequence Diagram

Deployment diagram (implementation)

DEPLOYMENT DIAGRAM (IMPLEMENTATION)

Example: Deployment Diagram for an Internet Based System

- A deployment diagram shows the placement of various **physical nodes** (components) across **different locations**
- A node can be thought of as a computer, or a bank of computers, representing a single computing resource



Satzinger (2011)

Component Diagrams vs. Deployment Diagrams

Component Diagrams

- Component diagrams focus on **logical** components
- Component diagrams show the various **executable components** of the new system and **how they relate** to each other

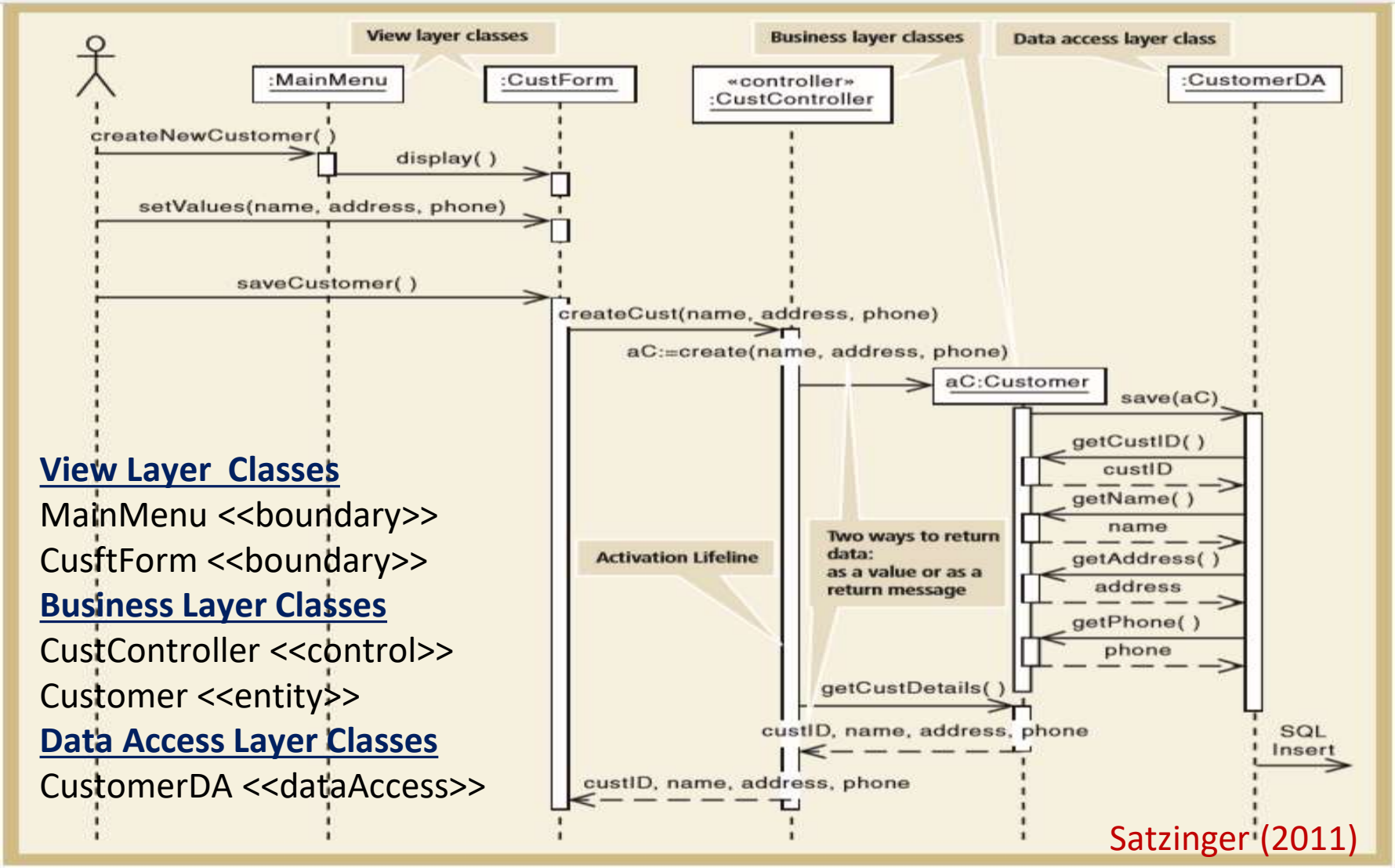
Deployment Diagrams

- deployment diagrams focus on **physical** components
- Deployment diagrams show **how and where** the components are executed on various computing platforms. Together they define the system's configuration

Objective 4:

USE CASE REALIZATION

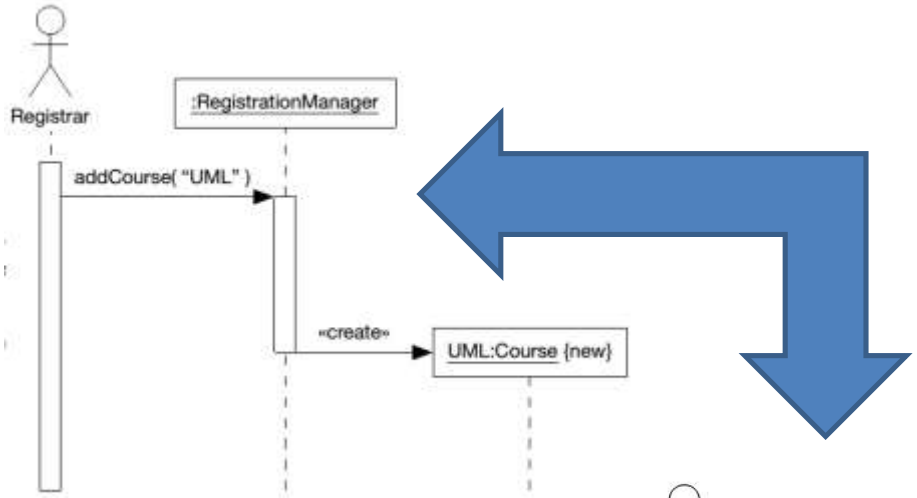
Use Case Realization: Example of a Complete Design Sequence Diagram



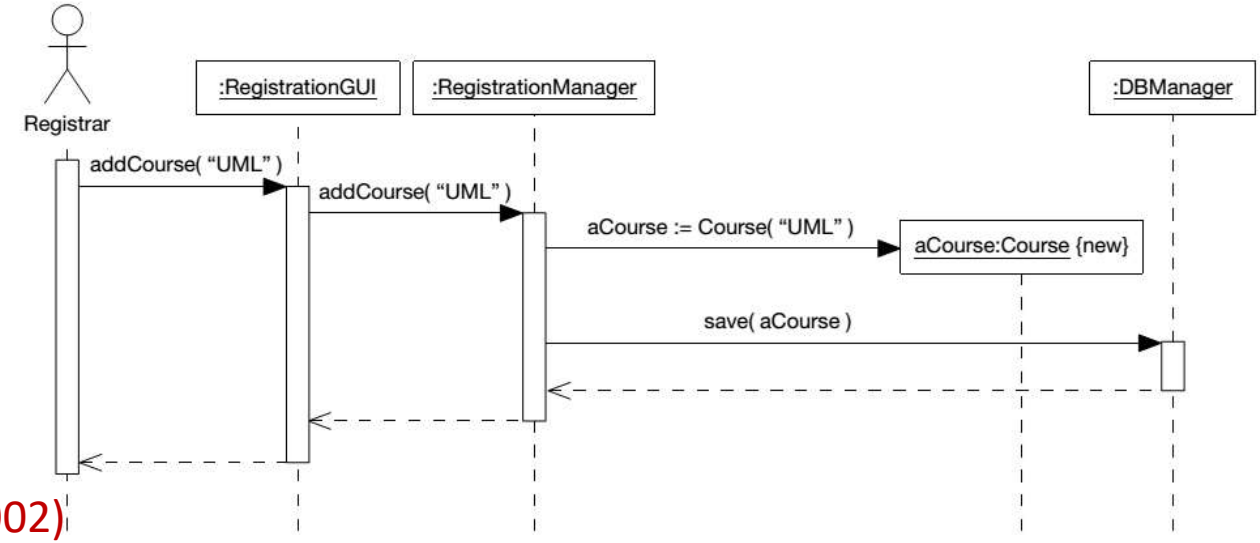
Steps for Use Case Realization: Three Refinement Steps

- **Step 1:** Add <<boundary>> classes in sequence diagram for **View Layer** **(already done in analysis but can be added in design)**
- **Step 2:** Add <<control>> class in sequence diagram for **Business Layer** **(already done in analysis but can be added in design)**
- **Step 3:** Add <<dataAccess>> class in sequence diagram for **Data Access Layer**

Step 1: Add <<boundary>> classes in sequence diagram



- GUI layer has been added
- Object construction is also shown by an explicit constructor method invocation.



Source: Arlow and Neustadt (2002)

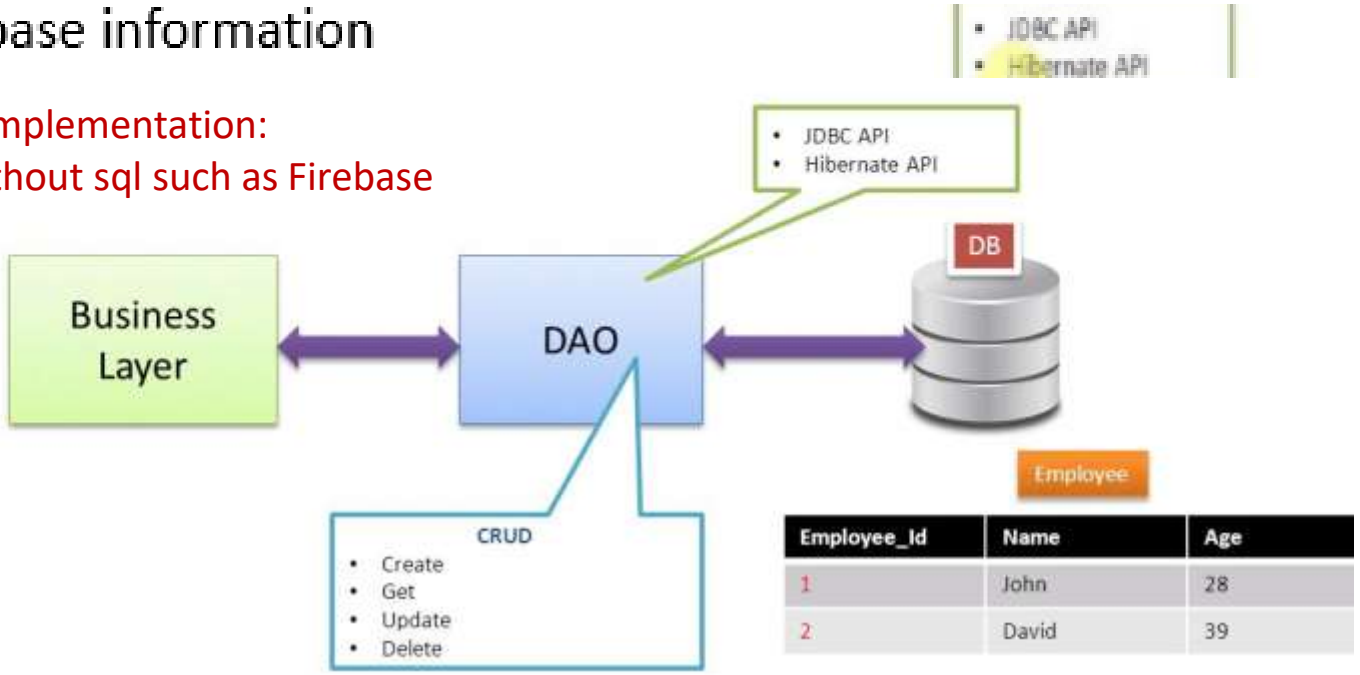
Step 2: Add <<control>> class in sequence diagram

- An artifact invented by the designer to **handle a system function**
  - Serves as a collection point for incoming messages
  - Intermediary between the outside world and the internal system
- A **single** use case controller results in **low cohesion**
- **Several** use case controllers raise coupling but result in **high cohesion**

Step 3: Add <<dataAccess>> class in sequence diagram

- The purpose of DAO is for:
  - Easier to change database without changing other classes
  - Security where only certain classes (DAO) can access the database information

*Note: Example of implementation:
- Cloud database without sql such as Firebase
- JDBC



Source: https://ramj2ee.blogspot.my/2013/08/data-access-object-design-pattern-or.html#.WtrBjIhuaDJ

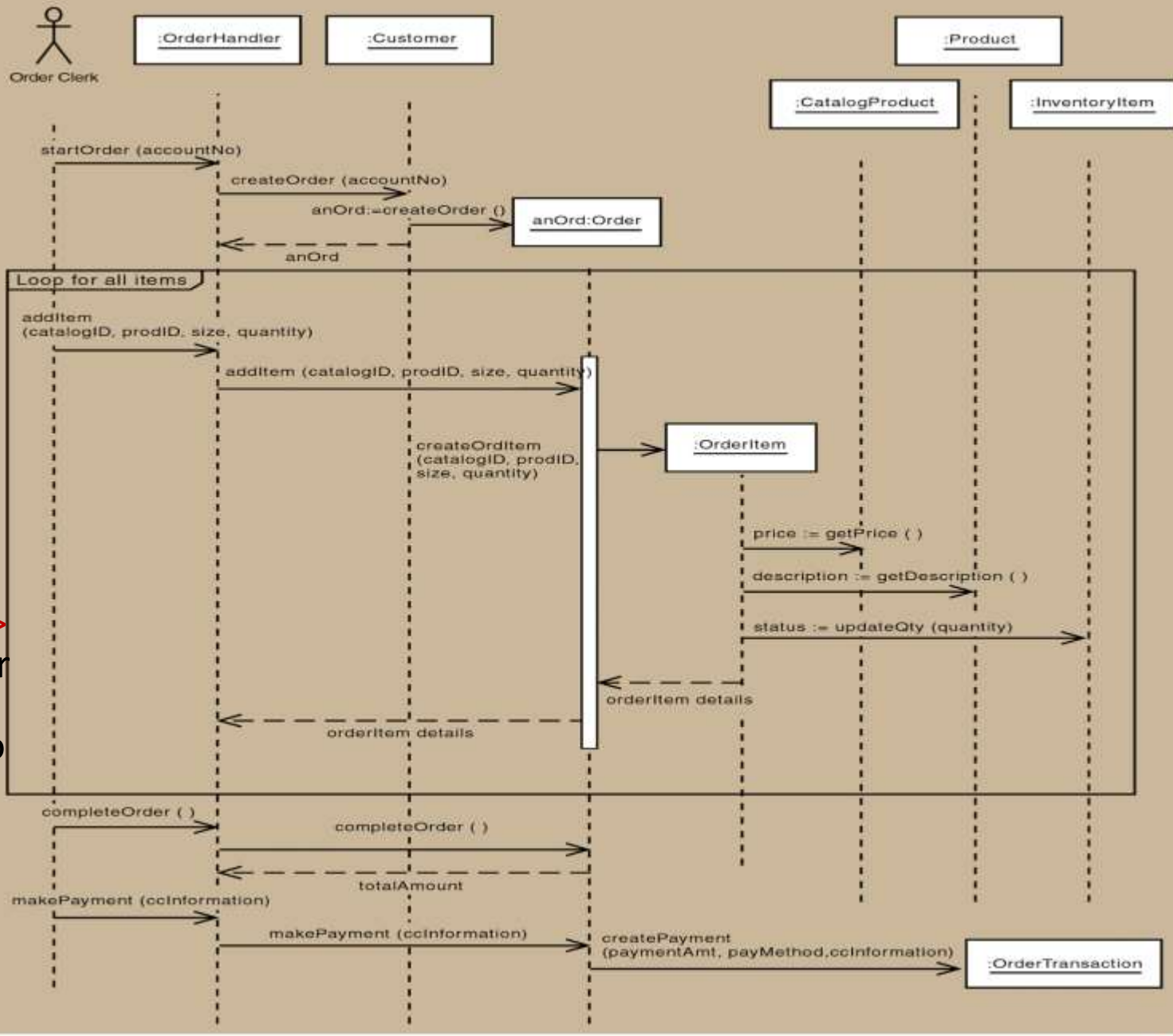
Example: Sequence Diagram for Telephone Order

- Define a user controller object
- Define a "create" message for new **Order** objects
  - Customer object creates the **Order** object
- Define other messages
  - **addItem, createOrdItem, getDescription, getPrice, updateQty**
- Identify source, destination, and navigation visibility for each message

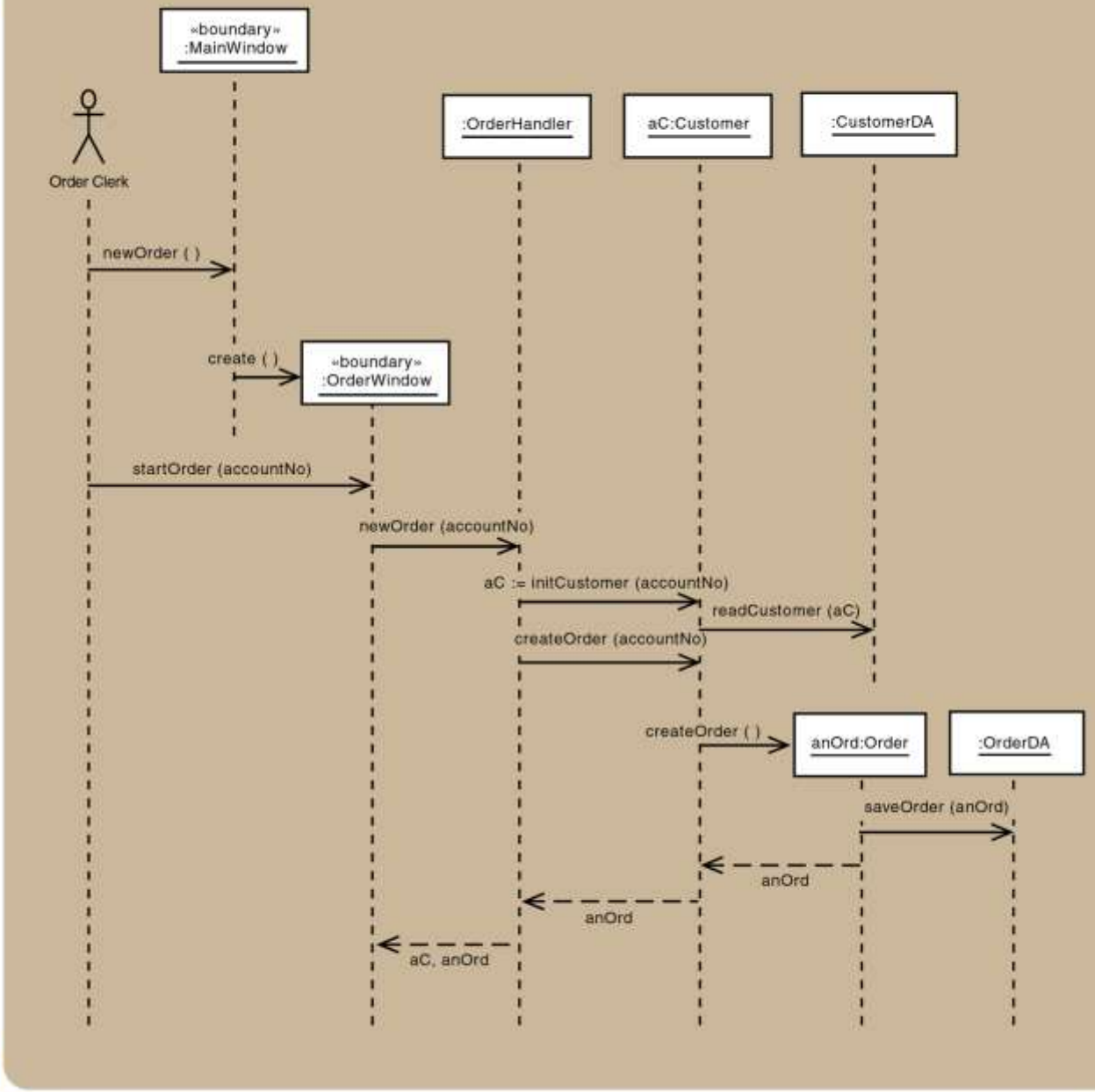
Developing a Multilayer Design for the Telephone Order Scenario

- Extend one message at a time
- **View layer**
  - Open Order window and return a Customer object
- **Data layer**
  - Customer object initializes itself
  - Add items to an order with a repeating message
  - Save Order and OrderItem to the database
  - Update database inventory
  - Complete transaction

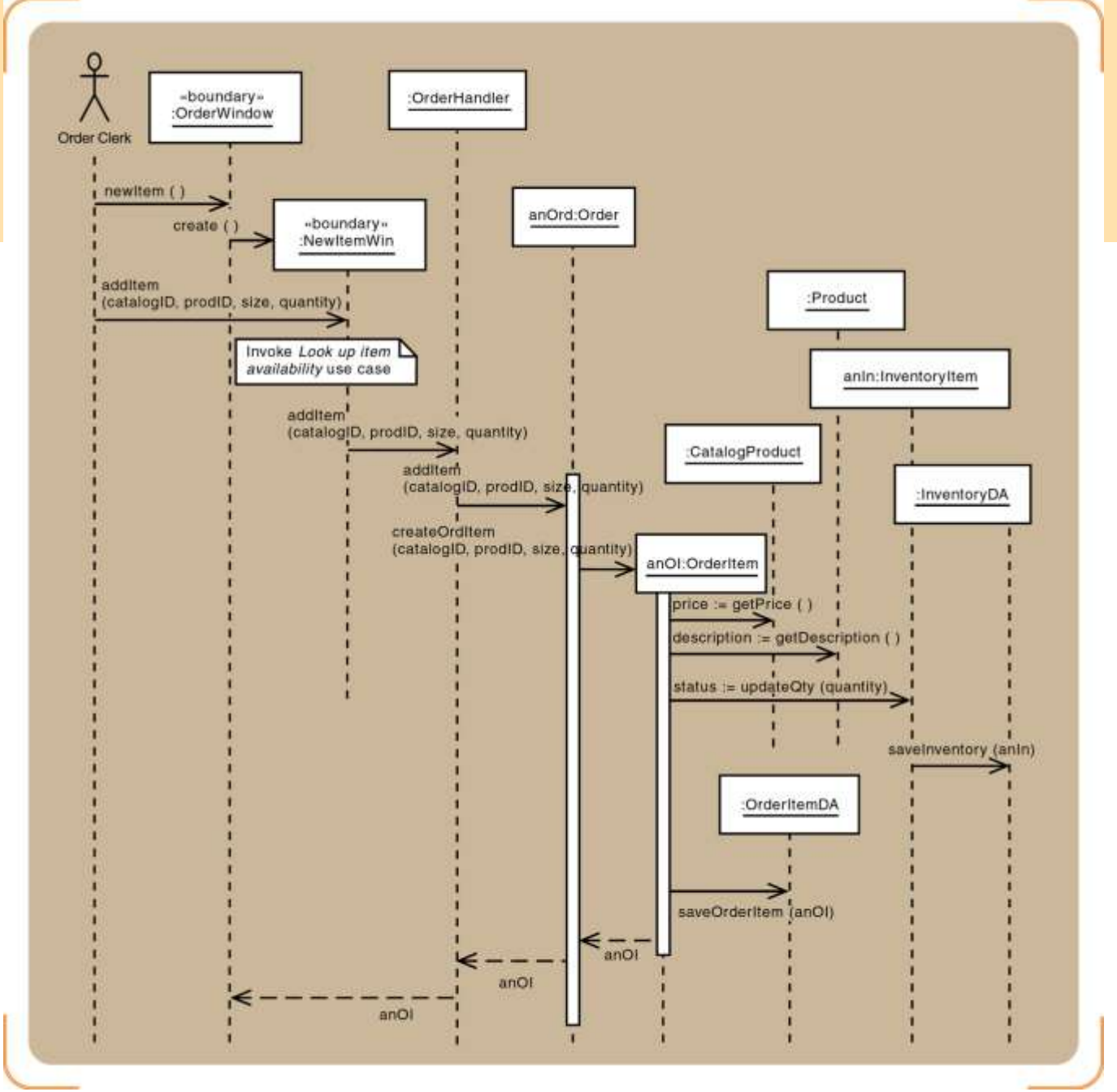
Sequence diagram with <<control>> and <<entity>> classes only for **Telephone Order** scenario



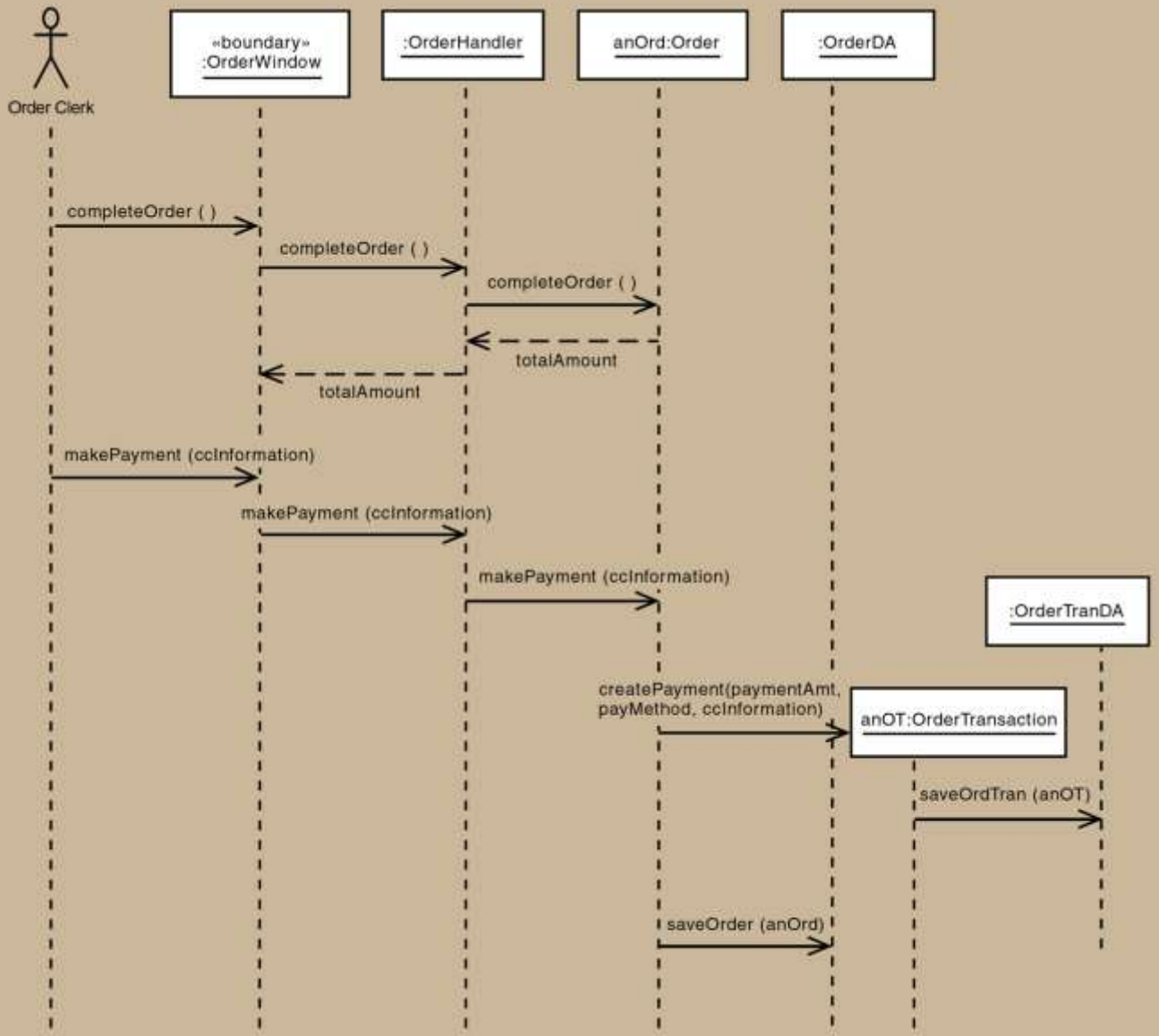
Telephone order sequence diagram for **newOrder** and **startOrder** messages - <<boundary>> and <<dataAccess>> classes are added



Telephone order sequence diagram for **newItem** and **addItem** messages - <<boundary>> and <<dataAccess>> classes are added

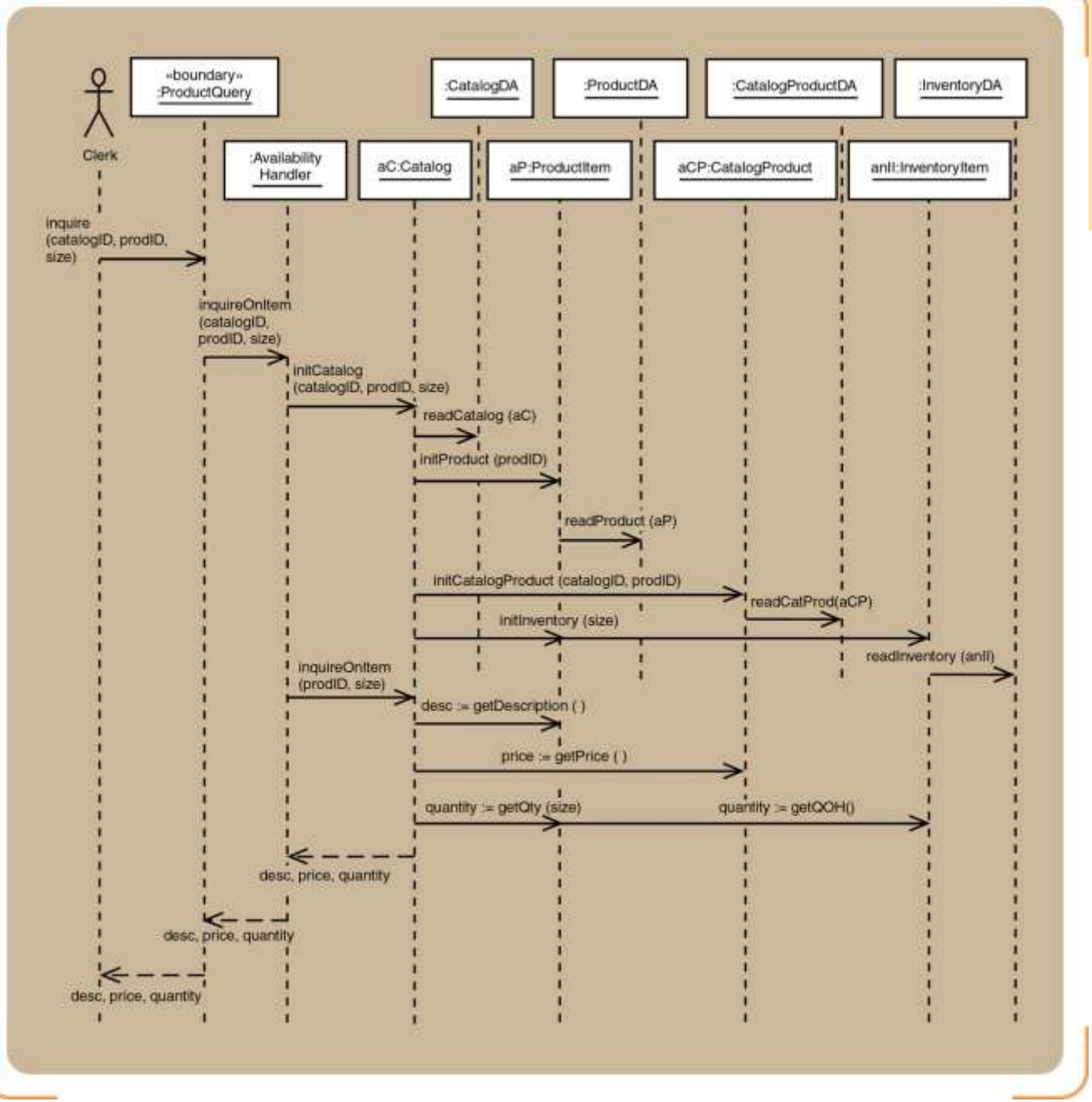


Telephone order sequence diagram for the final messages (**complete Order** and **makePayment**) - <<boundary>> and <<dataAccess>> classes are added



Design sequence diagram after the 3 refinement steps

Example: **Completed** three-layer design that includes <<boundary>>, <<control>>, <<entity>> and <<dataAccess>> classes for **Look up item availability** scenario

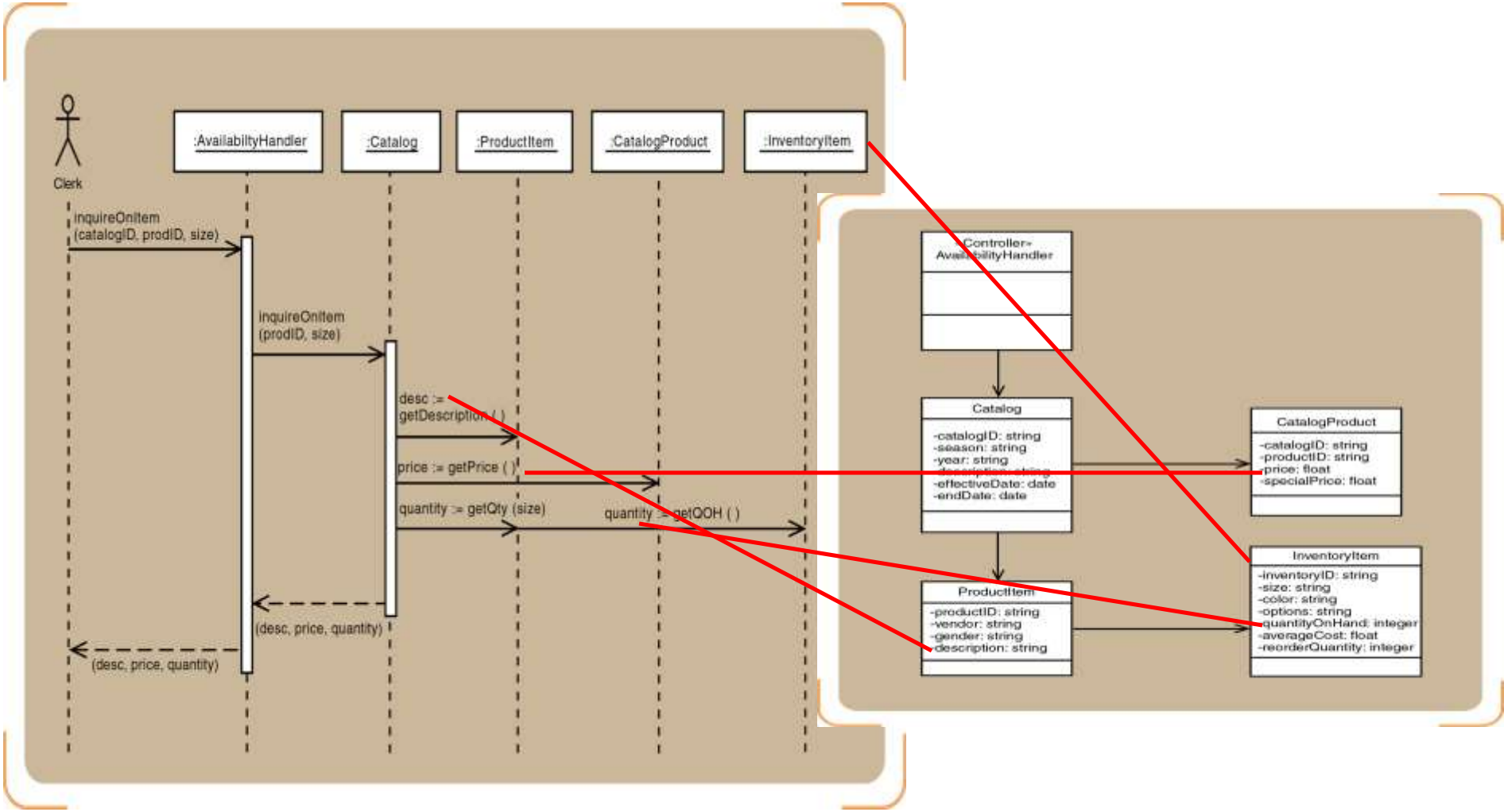


UPDATE DESIGN CLASS DIAGRAM

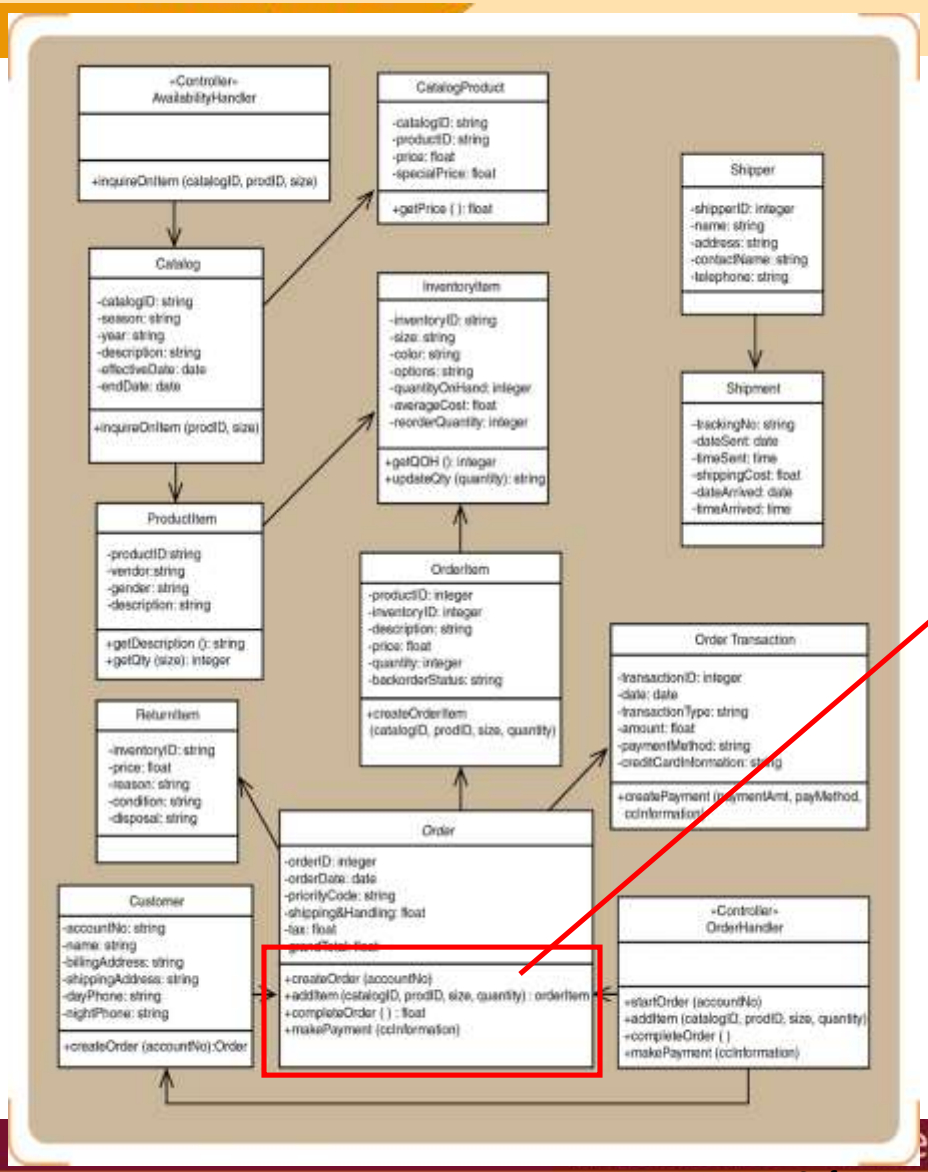
Updating Design Class Diagram

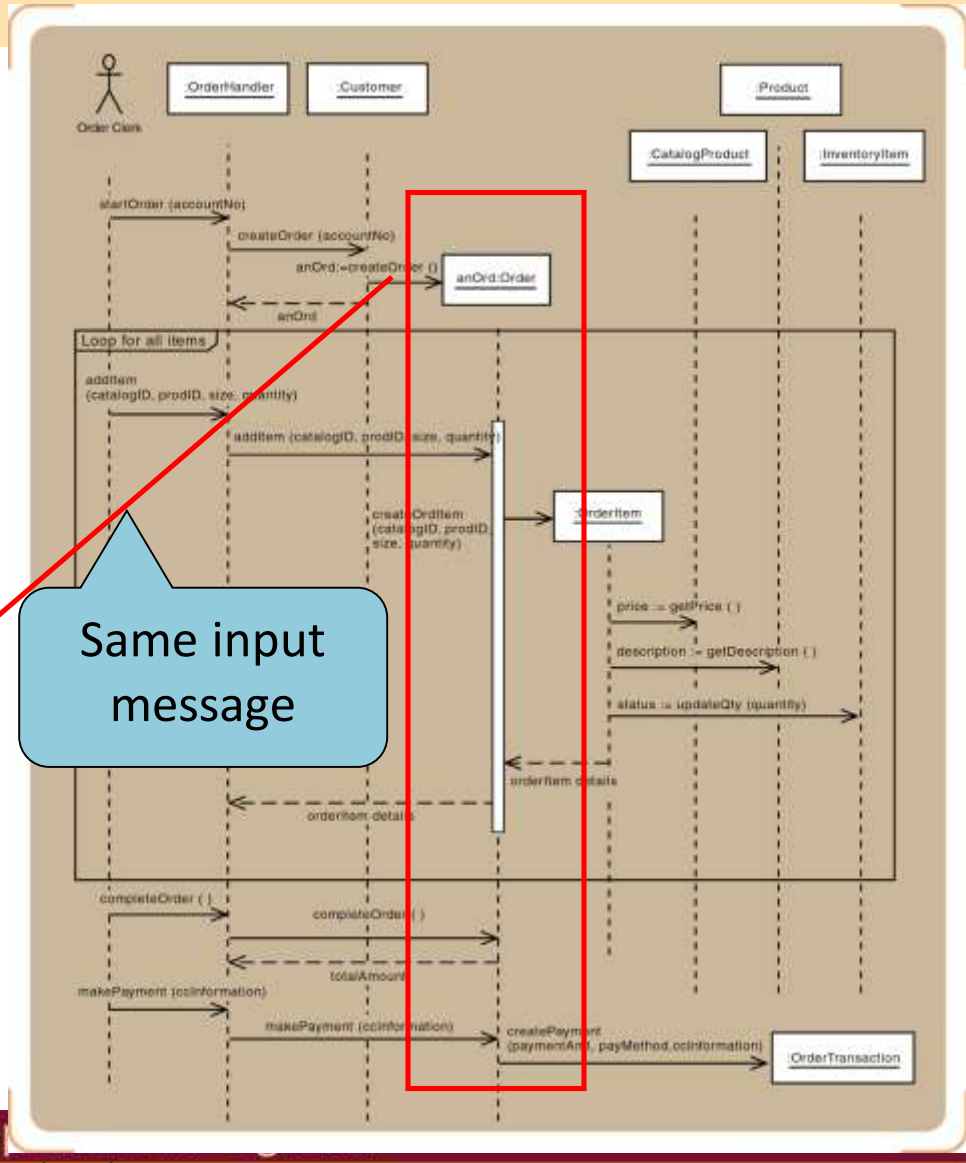
- Add classes for the **view and data access layers**
- Update classes with **method signatures**:
  - Constructor and get and set methods are optional
  - Use case specific methods are required
- Every **message** in a sequence diagram **requires a method** in the destination object

Sequence Diagram vs. Domain Model



Design Class Diagram vs. Sequence Diagram





Data Design

- Data design *"transforms the information domain model created during analysis into the data structures that will be required to implement the software."*

- Take the high-level data model from requirements and refine it into the concrete schema or data structures that the system will use.

https://www.testrail.com/blog/requirements-traceability-matrix/

Why Data Design is Important

- **Structure and Consistency:**
  A single source of truth eliminates ambiguity.
- **Implementation and Traceability:**
  A clear, traceable path from requirement to code.
- **Performance and Integrity:**
  An efficient foundation for a robust system.
- **Clarity and Alignment:**
  A shared reference that accelerates the team.

Process of Data Design

- **Identify and Model Key Entities:** Begin by identifying the key **entities** or data objects in the system from the analysis models.

- **Define Attributes and Data Types (Data Dictionary):** For each entity identified, list out its **attributes** and details. This is where you create or update the **data dictionary.** A convenient way to do this is to prepare a table for each entity that includes all attributes.

Description of Entities

The singular name of your core data object or class (e.g., 'User', 'Product').

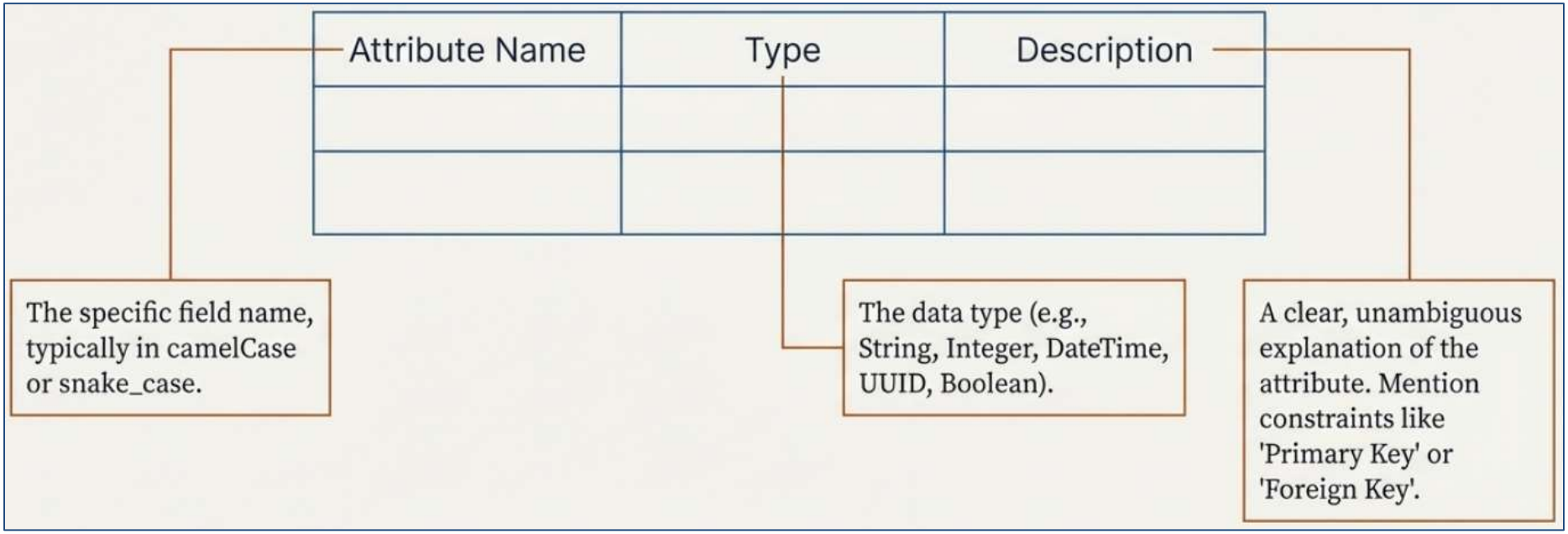
No.	Entity Name	Description

A concise, high-level summary of the entity's purpose and role in the system.

Example: Event Management System

No.	Entity Name	Description
1	Event	Represents a scheduled event, including its name, date, and topic.
2	Attendee	Represents an individual registered to attend one or more events.
3	Venue	Represents the physical location where an event is held.

Data Dictionary



Example: Event Management System (Event Entity)

Attribute Name	Type	Description
eventId	UUID	The unique identifier for the event. Primary Key.
eventName	String	The official public name of the event.
eventDate	DateTime	The scheduled start date and time of the event.
description	Text	A detailed description or abstract for the event.
venueId	UUID	The identifier for the location. Foreign Key to the `Venue` entity.

Traceability Matrix

- A traceability matrix (TM) is a table that maps and traces user requirements with corresponding elements such as:
  1. Requirement Identifier: A unique ID for each requirement.
  2. Description: A summary of the requirement.
  3. Design Artifacts: References to design documents or components that implement the requirement.
  4. Test Cases: IDs of test cases designed to verify the requirement.
  5. Test Results/Status: The outcome or status of the tests (e.g., pass/fail).
  6. Defects/Issues: Any issues or defects found during testing, linked back to the requirement.

- It ensures complete coverage so that all requirements are implemented, tested, and no unnecessary work is done.

Traceability Matrix in Plan-Driven

- An implementation of traceability matrix in plan-driven is a formal and comprehensive traceability document that is maintained throughout a project.

- It links requirements to design to implementation to testing, and finally to deployment according to the predefined project plan.

Example of Traceability Matrix in Plan-Driven

Package Item	Use Case ID	Use Case Description	Sequence Diagram ID	Sequence Diagram Description	Test Case ID
Package 1	UC-001	User Registration	SD-001	Register User	TC-001
					TC-002
					TC-003
	UC-002	User Login	SD-002	User Login	TC-004
					TC-005
	UC-003	Create New Post	SD-003	Create Post	TC-006
					TC-007
Package 2	UC-004	Update Profile	SD-004	Update Profile	TC-008
					TC-009
	UC-005	Search Posts	SD-005	Search Posts	TC-010
					TC-011

Agile Requirement Traceability Matrix (ARTM)

- The Requirements Traceability Matrix (RTM) is a tool or document commonly used to ensure that all the requirements established for a testing project are mapped to corresponding tests. Think of it as a management framework connecting requirements to design elements, tests, and outcomes.

https://www.testrail.com/blog/requirements-traceability-matrix/

How to create a Requirements Traceability Matrix

1. Define the project's scope and objectives
2. Gather and review requirements
3. Establish naming conventions
4. Set up your Requirements Traceability Matrix
5. Fill in the matrix

https://www.testrail.com/blog/requirements-traceability-matrix/

Example: ARTM

Table 7.1: Traceability

Sprint #	Subsystem/Package	User Story ID
	Package XX	US001
		US002
…		

An Example

- Login

Sprint #	Subsystem/Package	User Story ID
1	Login Interface	US001
		US002
…		

Key Points

- Object-oriented detailed design is a low level design involves the identification and description of sets of objects that must work together for each use case
- Fundamental design principles include encapsulation, object reuse, information hiding, navigation visibility, cohesion, coupling and separation of responsibilities
- Models involved in detailed design from analysis stage are class diagram and sequence diagram
- Classes with related resonsibilites should be grouped in a package as a subsystem of the system