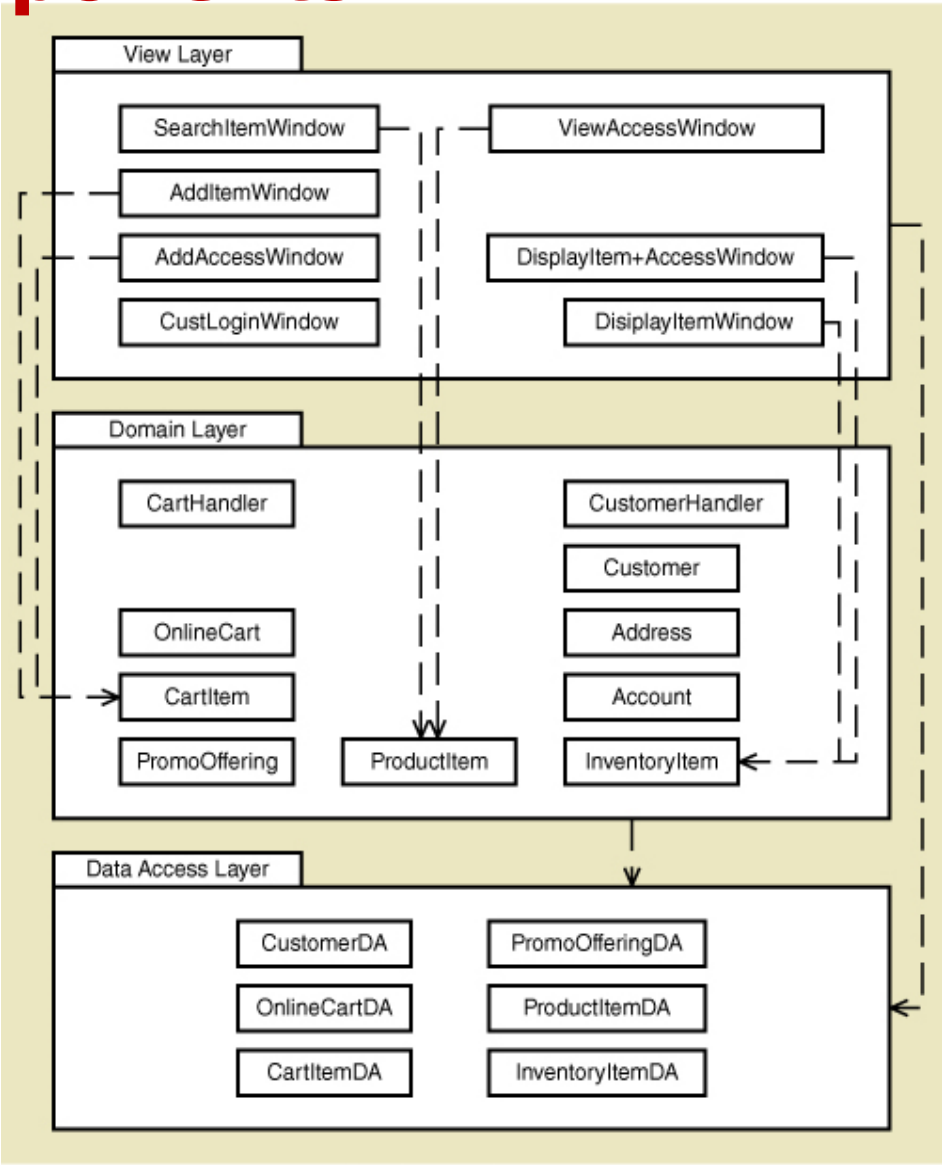
CSCI 4050/6050
Software Engineering

Object-Oriented Design

Fundamental Design principles

Dr. Eman Saleh eman.saleh@uga.edu

Typical models for defining application components



Package diagram

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

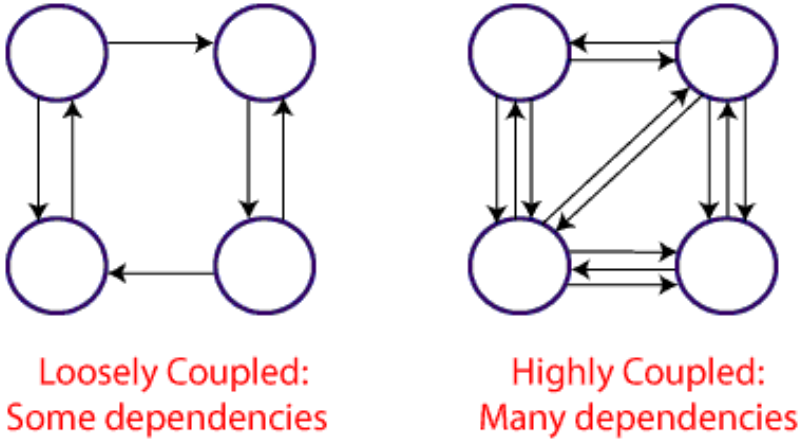
Fundamental Design Principles

- **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

```
import java.sql.*; // to save space
public class UserDA{
  public static void main (String[] args) {
     Connection conn = null;
     Statement st = null;
     ResultSet rs = null;
    try {
          Class.forName("com.mysql.jdbc.Driver");
          // Connection specifies the database schema ces,
          //user name in mySQL is root and the password is root
          conn =
          DriverManager.getConnection("jdbc:mysql://localhost:3306/ces?user=root&password=root");
          st = conn.createStatement();
          rs = st.executeQuery("SELECT * from user;");
         while (rs.next()) {
         String fname = rs.getString("firstname");
         String lname = rs.getString("lastname");
         int userID = rs.getInt("iduser");
         System.out.print ("UserID: " + userID+" ");
         System.out.print ("First name: " + fname+" ");
         System.out.println ("Last name: " + lname);
```

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

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

- **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.

Check class Listing!

What if we want to reuse class Listing in an application that compares prices but does need to store owner information.

What if a listing can be a vacant land.