PT821: Object-Oriented Programming

Lecture 2: Object-Oriented Design Concepts

Masoud Hamad

State University of Zanzibar
BITA Second Year - Semester 1

Lecture 2

Outline

1. Review of Previous Lecture
2. Understanding Object-Oriented Design
3. Identifying Objects and Classes
4. Relationships Between Objects
5. Encapsulation in Design
6. Designing Class Responsibilities
7. Practical Design Exercise
8. Design Guidelines and Best Practices
9. Next Steps

Quick Recap: OOP Fundamentals

Four Pillars of OOP:

1. Encapsulation
2. Inheritance
3. Polymorphism
4. Abstraction

Key Concepts:

- Classes vs Objects
- Real-world modeling
- Java as OOP language
- "Write once, run anywhere"

Today's Focus

We'll dive deeper into **Object-Oriented Design** - how to think in objects and design effective OOP solutions.

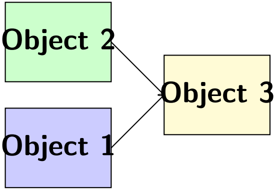
What is Object-Oriented Design?

Definition

Object-Oriented Design (OOD) is a software design methodology that models real-world entities as objects and defines interactions between these objects to solve problems.

Key Principles:

- Think in terms of objects
- Model real-world relationships
- Focus on responsibilities
- Design for reusability
- Minimize coupling
- Maximize cohesion



Design vs Programming

Design Phase

- **What** to build
- Identify objects and classes
- Define relationships
- Plan interactions
- Create blueprints
- Think before coding

Programming Phase

- **How** to build
- Write actual code
- Implement design
- Handle syntax
- Debug and test
- Optimize performance

Important

Good design leads to better code! Spending time on design saves debugging time later.

Finding Objects in the Real World

Step 1: Analyze the Problem Domain

Example

Problem: Design a university student management system

Noun Identification Technique:

- Read the problem statement
- Underline all nouns
- Identify which nouns are potential objects
- Group related nouns into classes

From our example:

”A university has students who are enrolled in courses. Each student has a name, ID number, and email. Courses have titles, codes, and instructors.”

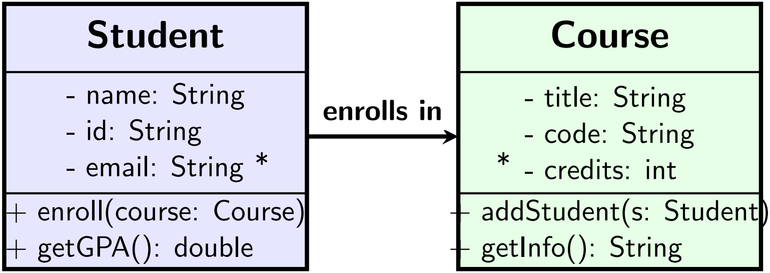
Potential Objects:

- University
- Student
- Course

Attributes:

- Name, ID, Email
- Title, Code

From Nouns to Classes



Class Structure:

- **Class Name** (top compartment)
- **Attributes/Fields** (middle compartment)
- **Methods/Operations** (bottom compartment)

Translating Design to Java Code

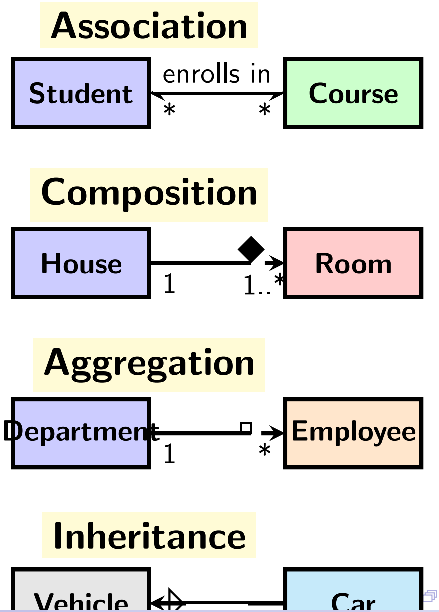
From UML to Java:

```
public class Student {
    // Attributes (private for encapsulation)
    private String name;
    private String id;
    private String email;

    // Constructor
    public Student(String name, String id, String
       email) {
        this.name = name;
        this.id = id;
        this.email = email;
    }

    // Methods
    public void enroll(Course course) {
        // Implementation here
        System.out.println(name + " enrolled in " +
```

Types of Object Relationships

1. **Association**
   - "uses" or "has a" relationship
   - Objects work together
   - Example: Student uses Library
2. **Composition**
   - Strong "part-of" relationship
   - Parts cannot exist without whole
   - Example: House has Rooms
3. **Aggregation**
   - Weak "part-of" relationship
   - Parts can exist independently
   - Example: Department has Employees
4. **Inheritance**
   - "is-a" relationship
   - Specialization/Generalization
   - Example: Car is a Vehicle

Relationship Types - Visual Representation



Association

Student enrolls in Course

* *

Composition

House Room

1 1..*

Aggregation

Department Employee

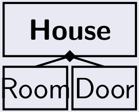
1 *

Inheritance

Vehicle Car

Composition vs Aggregation

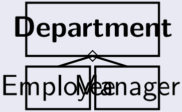
Composition (Strong)



Characteristics:

- Parts die with the whole
- Cannot exist independently
- Exclusive ownership

Aggregation (Weak)



Characteristics:

- Parts can exist independently
- Shared ownership possible
- Weaker relationship

Implementing Relationships in Java

Association Example:

```java
public class Student {
    private String name;
    private List<Course> enrolledCourses; // Association

    public void enrollInCourse(Course course) {
        enrolledCourses.add(course);
        course.addStudent(this);
    }
}

public class Course {
    private String title;
    private List<Student> students; // Bidirectional association

    public void addStudent(Student student) {
        students.add(student);
```

Designing for Encapsulation

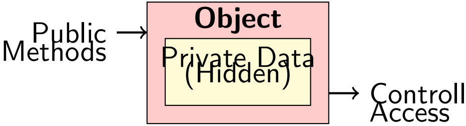
Encapsulation Principles:

Data Hiding:

- Make attributes private
- Provide public methods for access
- Control how data is modified
- Validate input data

Benefits:

- Prevents invalid states
- Easier maintenance
- Can change implementation
- Better debugging



Design Rule

Always ask: "Who needs to access this data and how?"

Good vs Bad Encapsulation

Bad Design

```
public class BankAccount {
    public double balance; // Public!
    public String pin;     // Exposed!

    // Anyone can change balance
    account.balance = -1000;
    account.pin = "1234";
}
```

Good Design

```
public class BankAccount {
    private double balance;
    private String pin;

    public void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
        }
    }

    public boolean withdraw(double amount) {
        if (amount > 0 && amount <= balance)
            {
            balance -= amount;
            return true;
        }
        return false;
    }

    public double getBalance() {
        return balance;
    }
}
```

Single Responsibility Principle

The Rule

Each class should have **one reason to change** - it should do one thing well.

Poor Design

- StudentManager class that:
  - Manages student data
  - Sends emails
  - Generates reports
  - Handles database connections
  - Calculates grades

Better Design

- Student class (data)
- EmailService class (communication)
- ReportGenerator class (reports)
- DatabaseManager class (persistence)
- GradeCalculator class (calculations)

Benefits:

Identifying Class Responsibilities

CRC Cards Technique (Class-Responsibility-Collaboration)

Student	
Responsibilities	**Collaborations**
- Store personal info	- Course (for enrollment)
- Track enrolled courses	- GradeBook (for grades)
- Calculate GPA	- ContactService
- Update contact details	- ValidationService

Questions to Ask:

- What does this class know? (Data/Attributes)
- What does this class do? (Methods/Behaviors)
- Who does it work with? (Other classes)

Design Challenge: Library Management System

Problem Statement: Design a library management system where:

- Patrons can borrow and return books
- Books have titles, authors, ISBN, and availability status
- Librarians can add new books and manage patron accounts
- The system tracks due dates and overdue fees
- Books can be reserved when unavailable

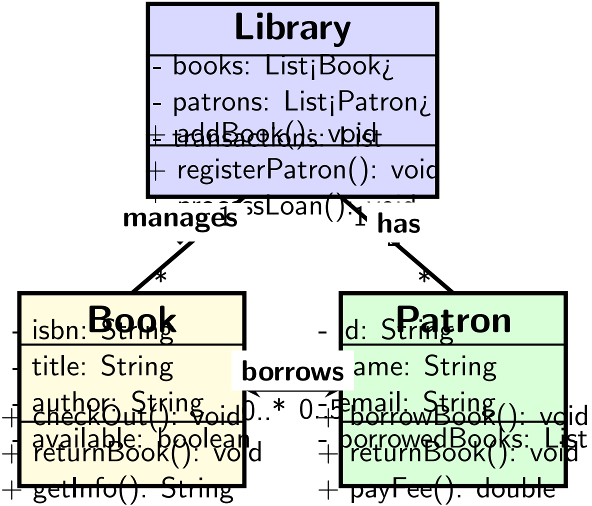
Your Task (5 minutes):

1. Identify the main objects/classes
2. List 3-4 attributes for each class
3. Define 2-3 key methods for each class
4. Identify one key relationship

Think-Pair-Share

Work with your neighbor to discuss your design choices!

Sample Solution: Library System



Library

- books: List¡Book¿
- patrons: List¡Patron¿
- transactions: List
+ addBook(): void
+ registerPatron(): voic
+ processLoan(): void

manages 1 *

has 1 *

Book

- isbn: String
- title: String
- author: String
- available: boolean
+ checkOut(): void
+ returnBook(): void
+ getInfo(): String

borrows 0..* 0..5

Patron

- id: String
- name: String
- email: String
- borrowedBooks: List
+ borrowBook(): void
+ returnBook(): void
+ payFee(): double

OOD Best Practices

Do's:

- Start with the problem domain
- Use meaningful class and method names
- Keep classes focused (SRP)
- Design for change and extension
- Favor composition over inheritance
- Program to interfaces

Don'ts:

- Don't create god classes
- Don't expose internal data
- Don't hardcode dependencies
- Don't ignore relationships
- Don't skip the design phase
- Don't fear refactoring

Design Mindset

Think objects, not procedures. Ask "What objects collaborate to solve this problem?" instead of "What steps do I need to follow?"

Common Design Mistakes

1. **Procedural Programming in OOP**
   - One class does everything
   - Methods that don't belong to objects
   - Data and methods in separate classes
2. **Poor Abstraction**
   - Too many public methods and attributes
   - Exposing implementation details
   - Classes with unclear purposes
3. **Tight Coupling**
   - Classes know too much about each other
   - Hard to change one without affecting others
   - Direct access to other classes' data

Remember

Good OOP design takes practice! Start simple and refactor as you learn.

Moving Forward: From Design to Implementation

Design Process Summary:

1. **Analyze** the problem domain
2. **Identify** objects and classes
3. **Define** attributes and methods
4. **Establish** relationships
5. **Apply** OOP principles
6. **Implement** in Java
7. **Test** and refine

Next Lecture Preview:

- Java class syntax in detail
- Constructors and method implementation
- Access modifiers (private, public, protected)
- Creating and using objects
- Practical coding exercises

Homework

Questions and Discussion

Any Questions About OOD?

Discussion Topics:

- Which relationships did you find in the library exercise?
- What makes a good class design?
- How do you decide what should be a separate class?

"Design is not just what it looks like and feels like. Design is how it works." - Steve Jobs

Thank You!

Next Class: Java Classes and Objects Implementation

Contact Information:
Instructor: Masoud Hamad
Email: massoud.hamad@suza.ac.tz
Office Hours: Thursday 08:00AM-12:00PM