FARABI UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE



Object-Oriented Analysis and Design

Lecture 6
Four Design Principles of Object-Oriented Software

Abstraction • Encapsulation • Decomposition • Generalization

Dr. Bazargul Matkerim, Professor-Researcher

Learning Outcomes

Explain

Distinguish the four principles
and their design purposes.

Apply

Build models with clear responsibilities and interfaces.

Evaluate

Check relationships, contracts
and trade-offs.

Use a university course-registration system as the running example.

Four Questions Guide the Design

Abstraction	Encapsulation	Decomposition
What details matter in this context?	How do we protect state behind a clear interface?	How do we divide work into coherent parts?

Generalization asks: what valid common concept can several types share?

1. Abstraction: Select the Essential



- Start with the purpose of the software.
- Keep relevant attributes and behaviours.
- Leave out detail that does not support that purpose.

A useful abstraction is selective, but still accurate enough for its intended task.

Context Changes the Model

Academic administration

Student ID
Degree programme
Course enrolments

Student club application

Club memberships
Interests
Participation in events

The same person can have different software representations in different contexts.

A Student Model for Registration

Relevant information	Relevant operations
Student ID Programme Current enrolments	Request enrolment View registered courses Withdraw from a course

“Attend a lecture” is real behaviour; model it only when the system needs to track it.

The Principle of Least Astonishment

Predictable interface

isEnrolled(course) returns
a status without changing enrolment.

Surprising interface

isEnrolled(course) silently
enrols the student as a side effect.

Names, results and side effects should match the expectations set by the interface.

Exercise: Choose a Cat Abstraction



- Context: a pet-care record application.
- Choose three useful attributes and two operations.
- Explain why fur texture may be irrelevant to this task.

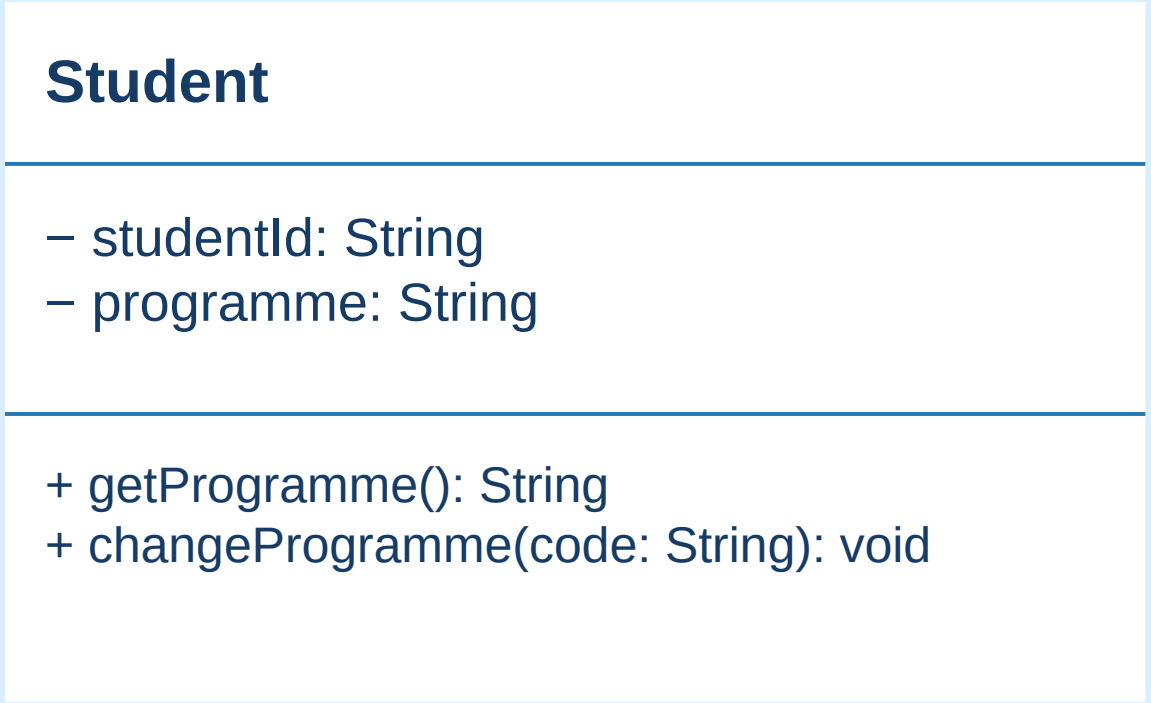
Choose software responsibilities, not every behaviour of the real animal.

One Possible Pet-Care Model

Attributes	Operations
Microchip ID	Record a feeding
Name	Update care instructions
Feeding schedule	View care history

A simulation might model eat() or sleep(); a care-record app mainly records information.

Abstraction in a UML Class



Name: the domain concept.

Attributes: selected information.

Operations: relevant behaviour.

This model exposes the concepts needed for registration; it does not describe every student detail.

2. Encapsulation: Control Access

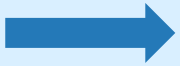
Bundle	Expose	Restrict
Keep related state and behaviour together.	Offer operations with clear contracts.	Hide details that callers do not need to manipulate.

Encapsulation enables a class to preserve its own rules when state changes.

The Interface Forms a Boundary

Caller

Requests enrolment



Public interface

enrol(studentId)
remainingSeats()

Hidden state

Enrolment collection
Capacity rule

Clients request behaviour through the interface instead of rewriting internal state.

Private State and Public Operations

Private state

capacity
registeredCount

Invariant: 0 ≤ count ≤ capacity

Public operations

enrol()
remainingSeats()

Reject enrolment when the class is full.

Access modifiers help enforce a boundary; the operations must still enforce the rules.

Java: Preserve the Capacity Rule

```
class CourseOffering {
  private int count = 0;
  private final int capacity;
  CourseOffering(int capacity) {
    if (capacity < 0) throw new IllegalArgumentException();
    this.capacity = capacity;
  }
  public boolean enrol() {
    if (count >= capacity) return false;
    count++;
    return true;
  }
}
```

Validate construction

Reject a full offering

Change state internally

Simplified example: duplicate students, withdrawals and concurrent requests are omitted.

Black-Box Thinking

What the caller needs	What can remain hidden
Operation name	Storage representation
Input requirements	Algorithm details
Result, effects and failures	Internal helper methods

The caller relies on the contract. Performance and observable effects still matter.

An Implementation Can Change

Version A

Store enrolments in a list.

Public query:
isEnrolled(studentId)

Version B

Store enrolments in a set.

Preserve the same query contract.

A stable interface can reduce changes in callers when representation changes.

Encapsulation Has Limits

It helps with	It does not replace
Maintaining valid state	Authentication and authorization
Localizing related changes	Encryption or access policies
Reducing direct dependencies	Correctness and concurrency tests

A private field is a programming boundary, not a complete security mechanism.

Exercise: Who Owns the Information?

Information to assign	Candidate classes
Student’s degree programme	Student
Students in a course offering	CourseOffering
Courses taught by a professor	Professor

Explain the responsibility that justifies each assignment; alternative models may be valid.

A Possible Responsibility Assignment

Student	CourseOffering	Professor
Knows the student's degree programme.	Tracks enrolments for a particular scheduled offering.	Knows assigned teaching responsibilities.

The university-wide course catalogue belongs in a separate catalogue responsibility.

3. Decomposition: Divide the Work



- Identify the responsibilities of the whole.
- Assign each coherent responsibility to a part.
- Define how the parts cooperate.

Decomposition breaks a problem into manageable parts; composition assembles them.

Refrigerator: Parts with Purposes

Cabinet and doors	Cooling system	Shelves and drawers
Enclose the storage space and limit heat exchange.	Remove heat and maintain the desired temperature.	Organize stored items and provide access.

The example illustrates responsibilities; a software design need not copy physical structure.

Decomposing Course Registration

RegistrationForm	RegistrationService	CourseOffering
Collect the request and show the outcome.	Check eligibility and coordinate enrolment.	Track enrolments and enforce seat limits.

Data access can be delegated to a repository when persistence is required.

Cohesion: Keep Related Work Together

Focused class	Unfocused class
CourseOffering manages capacity and enrolments. Its operations serve one role.	UniversityManager handles enrolment, payroll, printing and building maintenance.

High cohesion means the responsibilities of a module belong together.

Coupling: Manage Dependencies

Tight dependency

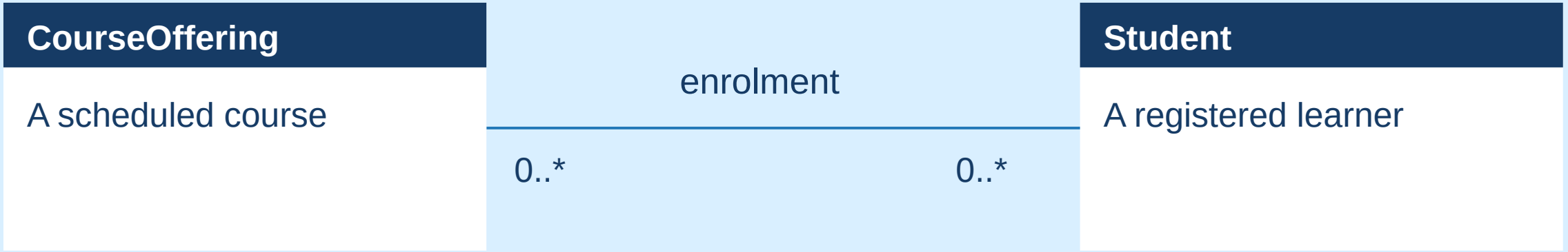
A form reads and edits
another object’s internal list.
A representation change breaks it.

Explicit interface

A form calls enrol(studentId)
and handles the documented result.
Internal storage can change.

Aim for necessary, understandable dependencies; zero coupling is rarely useful.

Multiplicity: How Many Related Objects?



This simplified model allows each student to enrol in several offerings.

Multiplicity describes allowed counts. It does not say that the same objects remain forever.

Lifetime: Existence and Ownership

Independent lifetime

A Student can exist before and after an enrolment. Deleting an offering must not delete the student.

Owned information

An OrderLine may be owned by one Order in the model. Its lifetime can depend on that owner.

Decide ownership from domain rules rather than from a physical analogy alone.

Sharing: One Object, Several Users

Shared reference

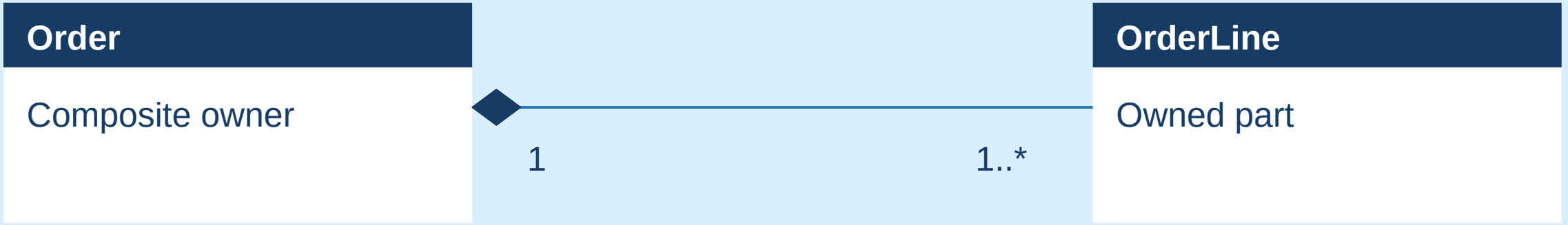
Several course offerings may
refer to the same Room.
The room is managed independently.

Exclusive ownership

An owned order line belongs
to at most one order at a time.
It is not simultaneously shared.

A relationship can be shared without making the related object a composition-owned part.

UML: Association and Composition



Assumption: a confirmed order owns one or more lines.
An ordinary association does not by itself imply lifetime ownership.

A filled diamond marks the composite owner. Composition implies exclusive ownership at a time.

Exercise: Model a Car Carefully



- Assume a conventional four-wheel car in operation.
- Which counts are fixed in this simplified model?
- Which related objects can change or exist independently?

State the scope. Passengers are related people, not composition-owned components.

Car Example: Suggested Answers

Counts under the assumption	Identity and lifetime
Four road wheels One steering wheel Zero or more passengers (up to the seat limit)	A wheel may be replaced without changing the count. A passenger exists independently of the car.

A fixed count does not imply fixed identity or the same lifetime as the whole.

4. Generalization: Find a Valid Common Type

Common concept

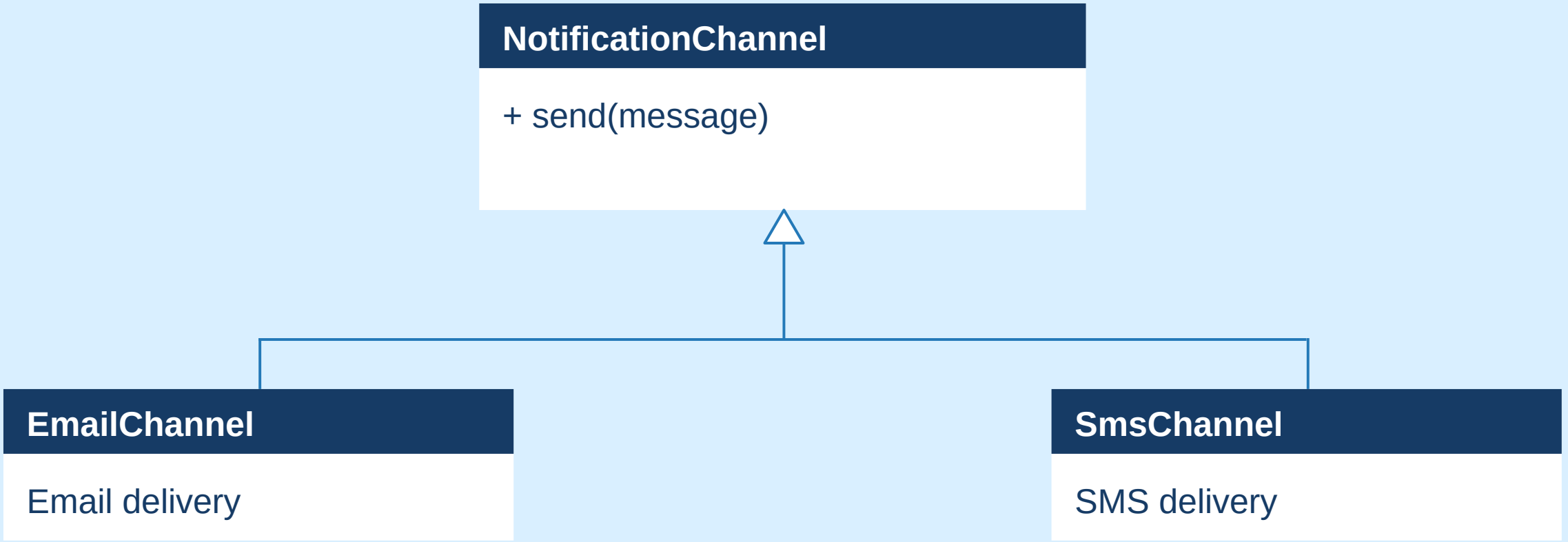
A NotificationChannel offers a send(message) operation.

Specialized types

EmailChannel and SmsChannel provide different implementations of that operation.

Generalization captures a valid “is-a” relationship, not merely similar-looking code.

UML Generalization



The hollow triangle points toward the more general classifier.

Substitutability Is the Key Test

Shared contract

send(message) attempts delivery.
It reports success or failure
according to the same rules.

Valid specialization

A caller can use EmailChannel
or SmsChannel without depending
on hidden type-specific rules.

A subtype should honour the expectations promised by its supertype.

Polymorphism: One Call, Different Behaviour

RegistrationService

Uses a NotificationChannel



Selected implementation

EmailChannel sends email.
SmsChannel sends SMS.

Common call: send(message)

Dynamic dispatch selects an implementation; the client depends on a common operation.

Inheritance and Composition

Inheritance: “is-a”

EmailChannel is a kind of NotificationChannel.

Use only when the shared contract remains valid.

Composition: “has-a”

RegistrationService uses a NotificationChannel.

The service delegates delivery to a collaborator.

Delegation often allows implementation changes without extending a class hierarchy.

Avoid Inheritance Just for Code Reuse

Tempting but misleading

Make Student extend Course
because both have an ID and name.

More meaningful design

Keep distinct domain concepts.
Share a helper only when it
represents a useful common concern.

Shared fields alone do not establish a valid subtype relationship.

Integrated Example: Register for a Course

RegistrationForm	RegistrationService	CourseOffering
Collect request	Coordinate checks	Enforce capacity

NotificationChannel
Send an outcome through a replaceable delivery implementation.

One scenario can reveal where each principle contributes to the design.

How the Four Principles Work Together

Abstraction and encapsulation

Model enrolment rather than
every aspect of campus life.
Protect capacity through operations.

Decomposition and generalization

Separate UI, coordination and domain rules.
Use a common notification contract
for alternative delivery types.

The principles support one another; applying one does not remove the need for the others.

Design Workshop

Produce a model

Sketch Student, CourseOffering, RegistrationService and a notification collaborator.
Assign key responsibilities.

Test your reasoning

Show a successful enrolment.
Show a full course offering.
Replace email with SMS.
Explain counts and lifetimes.

Deliverable: one class sketch and a short interaction walkthrough with stated assumptions.

Review: Four Design Questions

Select and protect

Abstraction: what matters here?
Encapsulation: what must callers use rather than manipulate directly?

Divide and generalize

Decomposition: who owns each task?
Generalization: which types share a valid contract?

Exit task: identify one design improvement and explain which principle supports it.