FARABI UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE



Object-Oriented Analysis and Design

Lecture 12
Course Summary and Case Studies

Dr. Bazargul Matkerim, Professor-Researcher

Learning Outcomes

Integrate the course

Translate requirements into responsibilities and collaborations.

Explain UML structure and its relationship to Java code.

Defend a design

Choose relationships and interface boundaries.

Use realistic changes and tests to evaluate cohesion, coupling and correctness.



Course Map

Analysis and modeling

Object-oriented thinking
Design within the software process
CRC cards and model evolution
Abstraction and encapsulation

Structure and design quality

Decomposition and generalization
UML classes and Java implementations
Interface contracts and polymorphism
Coupling, cohesion and separation of concerns



Analysis and Design

Analysis: understand the problem

Who uses the system?
What outcome do they need?
Which business rules must hold?
What failures matter?

Design: organize a solution

Which objects own each responsibility?
How do they collaborate?
What contracts cross the boundaries?
Which details should remain hidden?

CRC Cards as a Design Tool

Class and responsibilities

Name a domain concept or useful role. State what it knows and what it does.

Keep responsibilities focused on outcomes.

Collaborators

Identify who supplies information or performs delegated work.

Walk through a scenario and revise cards when a responsibility has no clear owner.



Core Object-Oriented Principles

Abstraction and encapsulation

Select concepts relevant to the problem.

Expose operations that protect meaningful state and invariants.

Decomposition and generalization

Split complex work into cohesive collaborators.

Use shared types when their behavioural contracts support substitution.



UML and Java: Quick Review

Class notation

− balance: int
+ deposit(amount: int): void

Visibility, names and types should agree with the intended implementation.

Relationship notation

Association: links between objects
Aggregation: shared whole–part meaning
Composition: exclusive lifecycle ownership
Generalization: solid hollow-triangle arrow
Realization: dashed hollow-triangle arrow

A Design Review Framework

Correctness

Are requirements and assumptions explicit?
Are multiplicities justified?
Do operations preserve invariants?
Are failures part of the contract?

Changeability

Are responsibilities cohesive?
Which implementation details leak?
Can a collaborator be replaced?
What does an expected change affect?



Case 1: University Library Lending

Requirements

A member borrows a physical book copy.
A copy has at most one active loan.
A member has at most five active loans.
Returning a copy closes its loan.

Scope and assumptions

Book describes a title or edition.
BookCopy represents a physical item.
Loan records a borrowing event.

Reservations and fines are outside this case.

Library Case: CRC Responsibilities

Domain objects

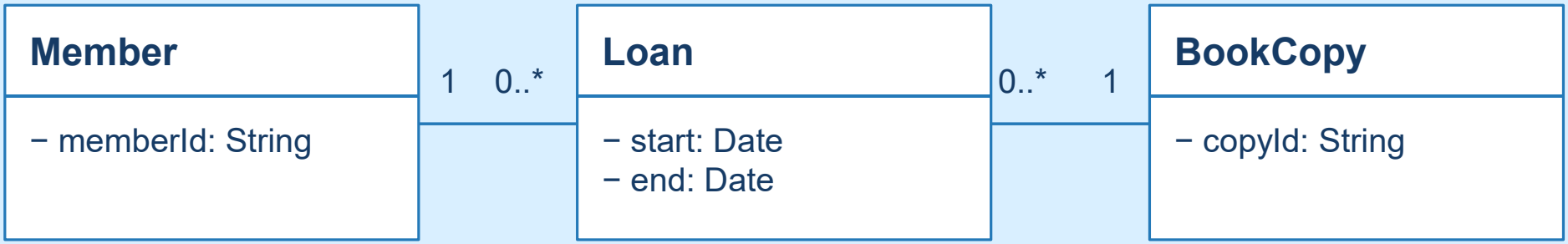
Member: identify the borrower.
BookCopy: identify a physical item.
Loan: record borrower, copy and dates.

Collaborators and coordination

LoanService: coordinate checkout and return.
LoanRepository: find and store loans.

The service checks lending rules within one transaction.

Library Case: Class Relationships



Active-loan limits are additional constraints

Each Loan links one Member and one BookCopy. History permits many loans, but only one active loan per copy.

Library Case: Checkout Collaboration

Checkout steps

Find the member and requested copy.
Check active-loan limits and availability.
Create and persist the loan.
Return its identifier and due date.

Consistency boundary

Concurrent requests must not both borrow the same copy.

Protect checking and creation with an appropriate transaction, lock or database constraint.



Library Case: Tests and Changes

Behavioural tests

Available copy creates one active loan.
Sixth active loan is rejected.
Already-borrowed copy is rejected.
Return closes the correct loan.

Change scenario

Staff may borrow more items than students.

Introduce a lending-policy role if rules vary.
Keep storage details outside that policy.

Case 2: Online Checkout

Requirements

An order contains item lines.
The system calculates a total and requests payment.
Failed payment leaves the order unpaid.

Explicit assumptions

One currency per order.
Amounts use integer minor units.
OrderLine belongs to one Order.
Payment processing is an external boundary.

Checkout Case: Mixed Responsibilities

Problematic CheckoutManager

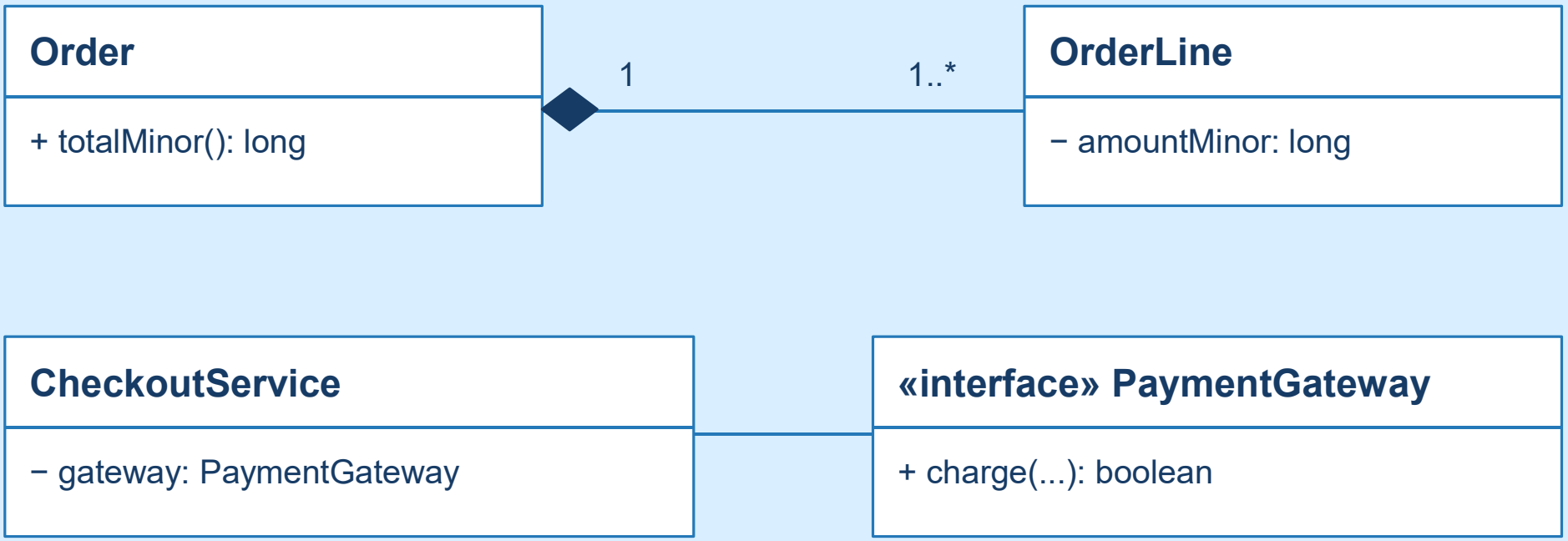
Calculates totals.
Builds bank-specific requests.
Updates database records.
Formats email receipts.

Design consequences

Pricing, bank integration and receipt formatting can change independently.

A large class makes those changes harder to isolate and test.

Checkout Case: Focused Contracts



Order owns its lines. CheckoutService coordinates payment through a gateway contract.

Checkout Case: Java Boundary

```
interface PaymentGateway {
  boolean charge(String orderId, long amountMinor);
}
class CheckoutService {
  private final PaymentGateway gateway;
  CheckoutService(PaymentGateway gateway) {
    this.gateway = java.util.Objects.requireNonNull(gateway);
  }
  boolean pay(Order order) {
    return gateway.charge(order.id(), order.totalMinor());
  }
}
```

Order provides id() and totalMinor().

A fake gateway controls test outcomes.

The boolean contract is simplified.

Production payment contracts need explicit outcomes, retry rules and idempotency. This is a classroom sketch.

Checkout Case: Failure and Retry Tests

Tests

Successful payment produces the paid outcome.
Rejected payment leaves the order unpaid.
A negative total is rejected.
A repeated request does not double-charge.

Design questions

Where is payment status stored?
How are uncertain outcomes reconciled?
Which identifier makes retries safe?

A class diagram alone cannot answer these.

Case 3: Smart Irrigation

Requirements

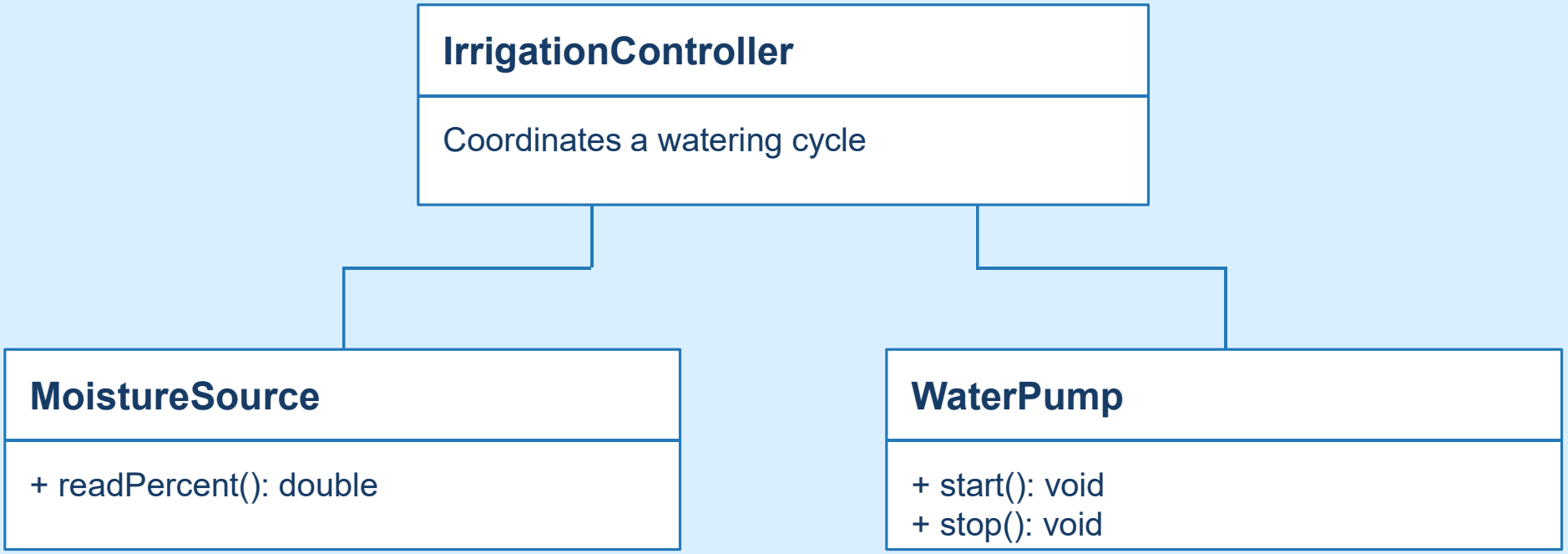
Read soil moisture.
Start watering below a configured threshold.
Stop after a bounded duration.
Handle missing sensor data safely.

Assumed policy

Readings use percent on a documented scale.
Missing or invalid input prevents a new watering cycle.

Thresholds and duration are configuration.

Irrigation Case: Responsibility Boundaries



The controller owns watering policy. Adapters translate sensor and pump protocols.

Irrigation Case: Policy Method

```
boolean shouldWater(double moisture, double threshold) {
  if (!Double.isFinite(moisture) || moisture < 0
      || moisture > 100) return false;
  if (!Double.isFinite(threshold) || threshold < 0
      || threshold > 100) {
    throw new IllegalArgumentException("threshold");
  }
  return moisture < threshold;
}
```

A pure decision is easy to test.

The threshold has explicit bounds.

Hardware access is outside this method.

Put this method inside a policy class. Sensor exceptions and pump timing require separate handling.



Irrigation Case: Boundary Tests

Input cases

Below threshold: allow watering.
Equal to threshold: do not water.
Above threshold: do not water.
NaN or out-of-range moisture: do not water.

System cases

Sensor failure prevents startup.
Timer stops an active cycle.
Pump failure is reported.

Replacing a sensor should affect its adapter, not the policy.

Case 4: Smartphone Components

Mixed implementation

One SmartPhone class contains image encoding and call-protocol logic.

Camera and calling changes affect the same implementation.

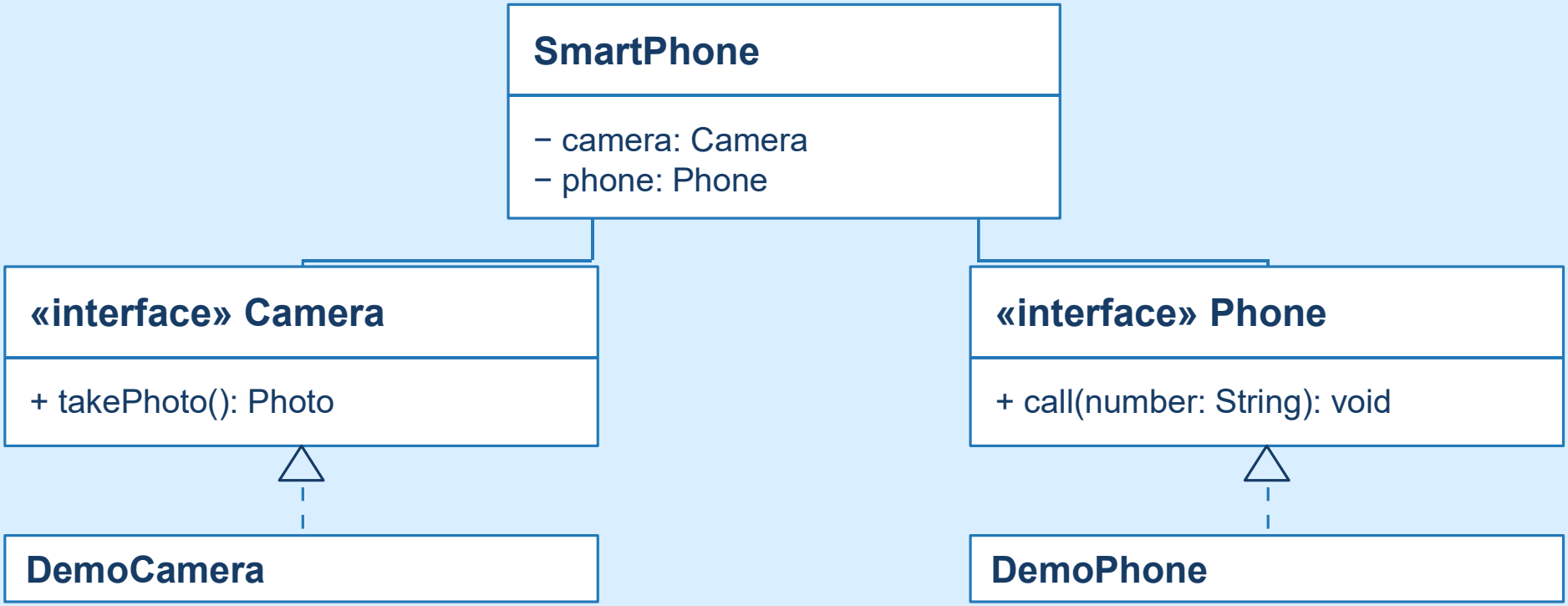
Separation of concerns

Camera handles image capture.
Phone handles calling.
SmartPhone coordinates both.

Interfaces describe the required capabilities.



Smartphone Case: UML Structure



Stored references use interfaces. Concrete components realize their respective contracts.

Smartphone Case: Substitution

Contract checks

A replacement camera returns the expected Photo type.
It preserves the documented failure contract.

Matching method signatures alone is insufficient.

Testing and assembly

Use a fake Camera to test delegation.
Choose concrete devices at assembly time.

SmartPhone can remain unchanged while components vary.



Comparing the Four Cases

Shared design decisions

Library: coordinate lending rules.
Checkout: isolate payment integration.
Irrigation: separate policy from hardware.
Smartphone: delegate distinct features.

Different risks

Library: concurrent borrowing.
Checkout: uncertain payment outcomes.
Irrigation: invalid data and stopping rules.
Smartphone: incompatible component contracts.



Capstone: Campus Equipment Booking

Scenario

Students reserve cameras and laptops.
Each physical item has a unique ID.
An item cannot have overlapping reservations.
Staff may approve restricted equipment.

Team deliverables

State assumptions and one use case.
Create CRC cards and a UML class diagram.
Define one replaceable service interface.
Sketch Java and propose failure tests.

Capstone Review Criteria

Model and behaviour

Distinguish equipment type from physical item.
Justify reservation multiplicities.
Explain the non-overlap rule.
Identify the concurrency boundary.

Design quality

Keep approval policy separate from storage.
Explain who coordinates booking.
Show how one expected change is contained.
Include a rejected-booking test.

Common Mistakes and Corrections

Modeling mistakes

Class for every noun: check responsibility first.
Diamond on every reference: justify ownership.
Inheritance for code reuse alone: check substitution.

Implementation mistakes

Setter for every field: protect invariants.
Interface with vendor details: clarify the contract.
Many tiny classes: check whether the boundary helps.

Final Synthesis

A repeatable workflow

Understand requirements and failure cases.
Assign responsibilities with CRC cards.
Model structure and contracts.
Implement, test and refine.

Exit challenge

Choose one case.
Explain its most important invariant.
Defend one class boundary.
Describe one change the design can absorb.

Good OOAD connects domain meaning, clear contracts and evidence from realistic scenarios.