FARABI UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE



Object-Oriented Analysis and Design

Lecture 10
Design principles 1

Coupling and cohesion in object-oriented design

Dr. Bazargul Matkerim, Professor-Researcher

Learning Outcomes

Evaluate a design

Explain coupling between modules and cohesion within a module.

Identify dependencies that make changes difficult.

Improve a design

Replace unclear interfaces with explicit contracts.

Refactor sensor and sprinkler examples while preserving behaviour.

Modules and Design Complexity

A module is a unit of software

A method, class, package or service can act as a module.

Choose the level of analysis before evaluating the design.

Complexity affects change

Ask how much a developer must understand to make a change safely.

Dependencies and mixed responsibilities can increase that burden.

Coupling and Cohesion

Coupling: between modules

How strongly does one module depend on other modules?

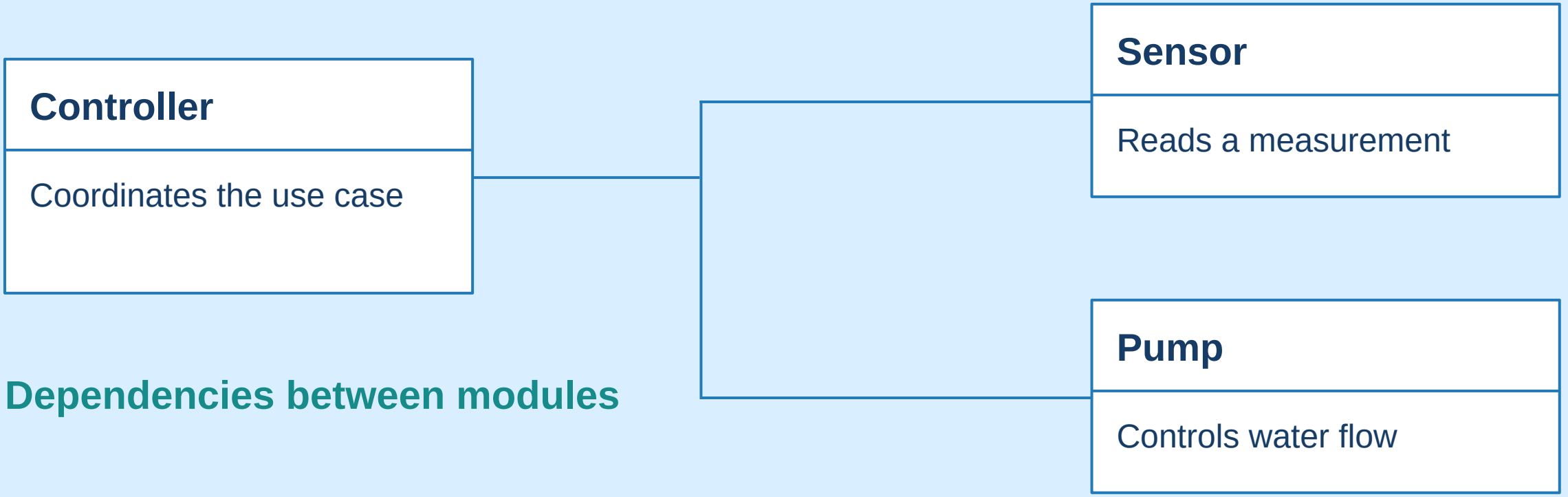
Consider the assumptions, protocols and implementation details it needs.

Cohesion: within a module

How closely do its responsibilities support a clear purpose?

Consider whether its data and operations belong together.

Dependency Structure



Dependencies between modules

More connections can increase coordination work. Dependency count alone does not establish design quality.

Change Propagation

A revealing question

If a sensor vendor changes its API, which modules need editing?

A dependency on vendor details can spread the change to many clients.

A useful boundary

An adapter can contain vendor-specific logic.

Clients depend on a stable measurement contract.

The adapter still needs maintenance.

Coupling: Degree

What to inspect

Number of collaborators
Amount of shared knowledge
Direction of dependencies
Cycles between modules

How to interpret it

A dependency on a stable value type differs from a dependency on an unstable service.

Use counts as review prompts, not universal quality scores.

Coupling: Ease of Use

An explicit contract

readCelsius()

The operation names the requested information and its unit.

Inputs, outputs and errors are documented.

Hidden obligations

get(1)

The caller must know a flag meaning. Perhaps it must also call init() first.

Such knowledge increases integration effort.

Coupling: Flexibility

Replacement question

Can another implementation satisfy the same contract?

Can a test double replace hardware during a test?

Limits of an abstraction

An interface helps only if it hides the details that vary.

A new implementation must preserve units, errors and behavioural promises.

Tight Coupling in Java

```
class TemperatureDisplay {
  private final VendorSensor sensor = new VendorSensor();

  public String text() {
    return sensor.readRaw() / 100.0 + " C";
  }
}
```

The display selects the vendor.

It knows the raw scale factor.

Testing requires control of that concrete sensor.

Illustrative API assumption: VendorSensor.readRaw() returns hundredths of a degree Celsius.

A Stable Measurement Contract

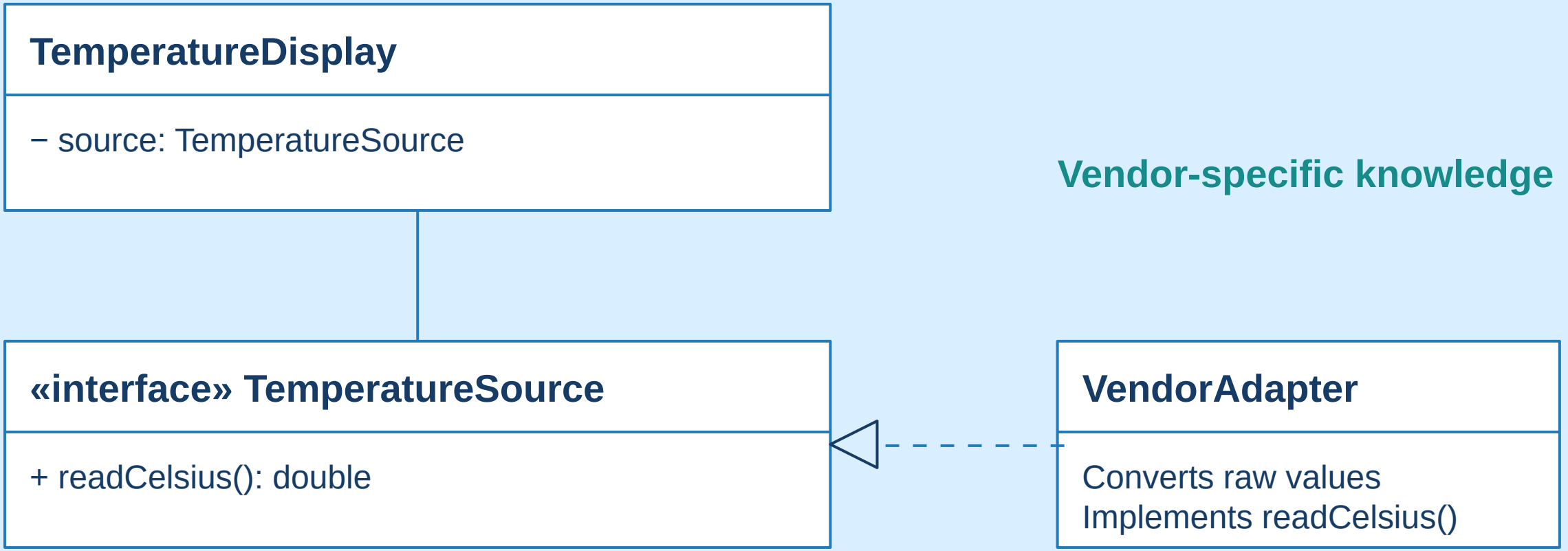
```
interface TemperatureSource {
  double readCelsius();
}
class TemperatureDisplay {
  private final TemperatureSource source;
  TemperatureDisplay(TemperatureSource source) {
    this.source = java.util.Objects.requireNonNull(source);
  }
  public String text() {
    return source.readCelsius() + " C";
  }
}
```

Constructor supplies the dependency.

The contract names the unit.

A lambda or fake can supply a test value.

Adapter and Client Responsibilities



The adapter translates the vendor API. The display formats a value. Each can change for a different reason.

High Cohesion

Related responsibilities

A class’s operations work together toward a clear purpose.

For a BankAccount, balance queries and valid deposits may belong together.

More than one method

High cohesion does not mean one method per class.

Operations can be cohesive when they share meaningful state and preserve the same invariants.

Low Cohesion

Mixed purposes

A ReportManager calculates totals, controls a printer and manages login passwords.

These responsibilities may change independently.

Review questions

Can the class’s purpose be stated clearly?
Do methods use largely unrelated data?
Do unrelated requirements repeatedly change the same class?

Cohesive Responsibility Boundaries

Possible separation

ReportCalculator computes report values.

ReportPrinter handles printing.

AuthenticationService verifies access.

A design decision

Separate responsibilities when they have distinct policies or reasons to change.

Avoid splitting a coherent operation into many classes without a benefit.

Sensor Example: A Flag-Based API

```
class Sensor {
  private double humidity;
  private double temperature;

  public double get(int controlFlag) {
    switch (controlFlag) {
      case 0: return humidity;
      case 1: return temperature;
      default: throw new IllegalArgumentException("flag");
    }
  }
}
```

Caller must know that 0 means humidity.

The returned unit depends on the flag.

Fields represent previously sampled values.

This corrected teaching example returns double. Acquisition of measurements is outside the snippet.

Sensor Example: Two Separate Questions

Coupling concern

Clients depend on numeric flag meanings and result units.

Adding another measurement may spread flag-handling logic across clients.

Cohesion concern

A combined climate sensor can have a coherent purpose.

Separate types are useful if measurements have independent hardware, validation or change patterns.

Option A: Explicit Queries

```
class ClimateReading {
  private final double humidityPercent;
  private final double temperatureCelsius;
  ClimateReading(double humidity, double temperature) {
    humidityPercent = humidity;
    temperatureCelsius = temperature;
  }
  public double humidityPercent() { return humidityPercent; }
  public double temperatureCelsius() { return temperatureCelsius; }
}
```

One coherent sampled reading.

Names expose units.

No numeric selector flag.

Assumption: acquisition validates the values before creating this immutable reading.

Option B: Separate Measurement Contracts

```
interface TemperatureSource {
  double readCelsius();
}

interface HumiditySource {
  double readPercent();
}
```

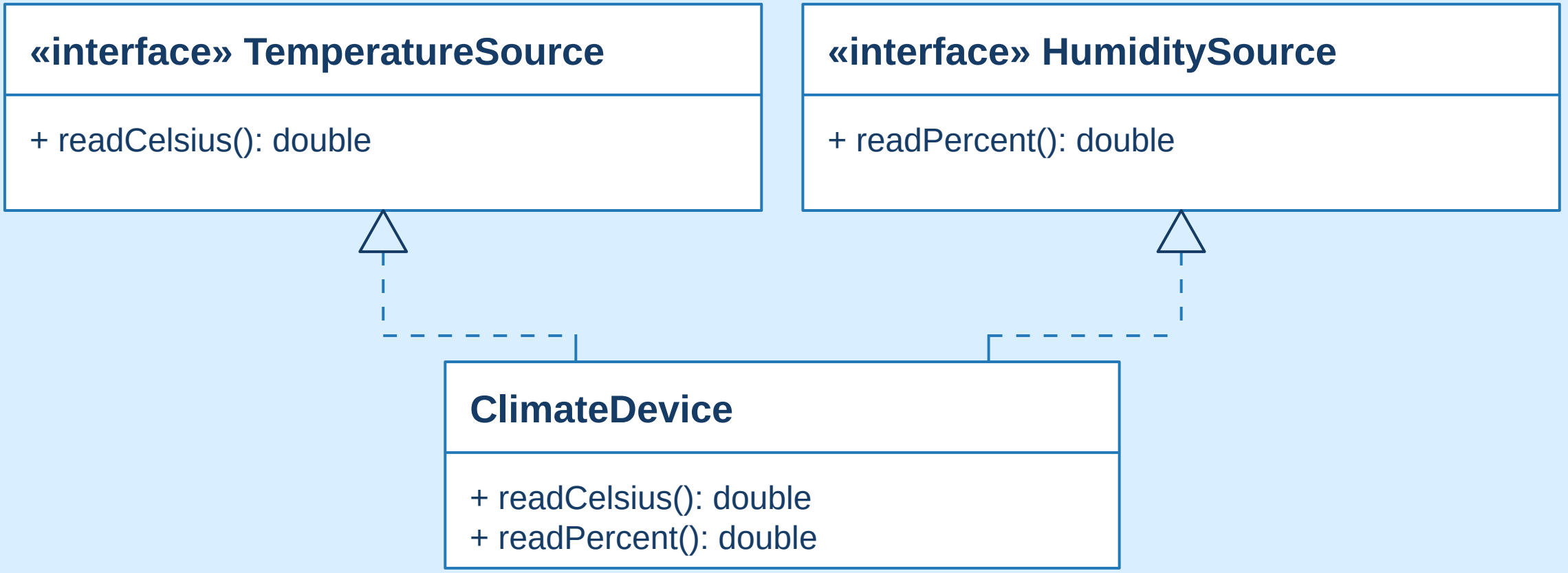
Clients request only the capability they need.

One physical device may implement both interfaces.

Separate classes are optional.

Define valid ranges, unavailable readings and failure behaviour in each contract.

A Device with Two Focused Interfaces



Dashed realization links show that one device satisfies two contracts. Clients can depend on either role.

Lawn Sprinkler: Mixed Policy and Hardware

```
class LawnSprinkler {
  private final VendorPump pump = new VendorPump();

  public void water() {
    pump.writeRegister(7, 1);
  }
}
```

The class coordinates watering.

It also knows a vendor register and command value.

A hardware change affects watering code.

Illustrative API assumption: writing value 1 to register 7 starts this vendor’s pump.

Delegation to a Pump Contract

```
interface WaterPump { void start(); }
class LawnSprinkler {
  private final WaterPump pump;
  LawnSprinkler(WaterPump pump) {
    this.pump = java.util.Objects.requireNonNull(pump);
  }
  public void water() { pump.start(); }
}
```

The pump adapter owns hardware commands.

LawnSprinkler delegates through start().

A fake pump supports isolated tests.

Partial model: a real watering cycle also needs stop control, timing and failure handling.

Balancing Cohesion and Coupling

Useful refactoring

Moving vendor commands into an adapter can improve both cohesion and coupling.

The goals are not a fixed zero-sum trade-off.

Costs to consider

More boundaries can introduce calls, configuration and coordination.

Prefer boundaries that isolate real variation and keep contracts understandable.

Workshop: Review an Irrigation Design

Scenario

IrrigationManager reads soil moisture, writes pump registers, formats dashboards and emails reports.

Propose responsibilities and dependency directions.

Evidence to provide

Draw a class diagram.
Identify one hardware adapter.
State units and failure behaviour.
Show how a fake sensor enables a test.
Explain one cost of your design.

Review and Design Checklist

Coupling questions

What does this module depend on?
Which details does it know?
How widely does a change spread?
Can a collaborator be replaced safely?

Cohesion questions

What is the module’s purpose?
Do its operations support that purpose?
Which responsibilities change independently?
Does a proposed split improve clarity?

Aim for focused responsibilities and explicit dependencies. Evaluate the design against realistic changes.