FARABI UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE



Object-Oriented Analysis and Design

Lecture 11
Design principles 2

Separation of concerns in object-oriented design

Dr. Bazargul Matkerim, Professor-Researcher

Learning Outcomes

Identify concerns

Recognize distinct responsibilities in a problem.

Explain how separation of concerns relates to cohesion and coupling.

Design collaborating classes

Assign responsibilities to suitable classes.

Use interfaces and delegation to separate smartphone features.

What Is a Concern?

Something the solution must address

A concern can be a capability, rule, constraint or quality requirement.

Examples: taking a photo, accepting payment, protecting user data.

Different levels of detail

A concern may contain smaller concerns.

Payment processing includes validation, authorization and failure handling.

The scope determines useful boundaries.

Separation of Concerns

Core idea

Organize the solution so that distinct concerns have identifiable places.

Each part exposes a contract and hides relevant implementation details.

Collaboration remains necessary

Separated parts still communicate.

Define who coordinates them and what information crosses each boundary.

Supermarket Analogy

Distinct responsibilities

Bakery prepares bread.
Stockroom replenishes shelves.
Checkout accepts payment.

Each role requires different knowledge.

Shared workflow

Inventory and sales connect these roles.

Separating work does not eliminate coordination, shared rules or handoffs.

Concerns in a Software System

User interface	Business rules
Accept input and present results	Apply domain policies

Persistence	Security
Store and retrieve information	Control access and protect data

These are responsibility areas. A small system need not create a separate service for every concern.

Connection to Object-Oriented Design

Abstraction and encapsulation

Abstraction selects relevant concepts and operations.

Encapsulation keeps implementation details behind a controlled interface.

Decomposition and generalization

Decomposition separates a whole into collaborating parts.

Generalization captures a shared contract when substitution makes sense.

Responsibility Assignment

Questions to ask

What outcome does this object provide?
What information does it need?
Which rules does it protect?
Which work should it delegate?

A useful boundary

Group behaviour that supports one coherent purpose.

Separate policies or mechanisms that change independently.

Dog Example: Eating and Providing Food

Dog responsibility

The dog knows how to eat supplied food.

The eating operation need not choose where the food comes from.

Owner responsibility

In this household scenario, the owner provides food.

This is a modeling assumption, not a claim that every dog depends on an owner.

Delegating the Eating Action

```
class Food { }
class Dog {
  public void eat(Food food) {
    java.util.Objects.requireNonNull(food);
    System.out.println("Eating");
  }
}
class DogOwner {
  public void feed(Dog dog, Food food) {
    dog.eat(food);
  }
}
```

Owner initiates feeding.

Dog performs eating.

Food is an input to the action.

Simplified model: dietary rules and food sourcing are outside this example. Assume dog is non-null.

Dog Example: Responsibility Review

What belongs together?

Dog: behaviour and state needed for eating.

DogOwner: deciding when and what to provide in this scenario.

What does the model imply?

Passing Food as a parameter does not establish permanent ownership.

Add stored associations or lifecycle rules only when the requirements need them.

Smartphone Example: Scope

Selected capabilities

Take a photo.
Make a phone call.

Other smartphone features remain outside this teaching model.

Reasons to separate them

Camera processing and call handling require different policies and hardware knowledge.

They can evolve independently.

One Class with Mixed Implementation

```
class SmartPhone {
  public void takePhoto() {
    // Configure sensor, capture pixels, encode image
  }
  public void call(String number) {
    // Validate number, connect radio, transmit audio
  }
}
```

Both mechanisms live in one class.

A camera change may affect the same class as a calling change.

Comments represent omitted implementations.

A façade may legitimately expose several features. The concern is mixing their implementation details.

Responsibility Map

Camera concern

Capture an image.
Handle camera-specific processing.
Report capture failures through its contract.

Phone concern

Start a call.
Handle calling-specific operations.
Report call failures through its contract.

SmartPhone coordinates these capabilities and exposes convenient operations to its clients.

Interfaces for the Capabilities

```
record Photo(String description) { }

interface Camera {
  Photo takePhoto();
}

interface Phone {
  void call(String number);
}
```

Camera returns a Photo.

Phone accepts a destination number.

The record is a placeholder result for the demonstration.

Examples use Java 17-compatible syntax. Production contracts must define valid inputs and failure behaviour.

Camera Implementation

```
class DemoCamera implements Camera {
  @Override
  public Photo takePhoto() {
    return new Photo("Demo image");
  }
}
```

The class realizes the Camera contract.

A real implementation owns capture and encoding details.

SmartPhone only needs takePhoto().

This implementation simulates capture. It does not access a physical camera.

Phone Implementation

```
class DemoPhone implements Phone {
  @Override
  public void call(String number) {
    if (number == null || number.isBlank()) {
      throw new IllegalArgumentException("number required");
    }
    System.out.println("Calling " + number);
  }
}
```

Validates the demonstration input rule.

Owns call-related behaviour.

Uses console output to simulate a call.

For this example, Phone.call requires a non-blank number. Real number validation needs a fuller specification.

SmartPhone as Coordinator

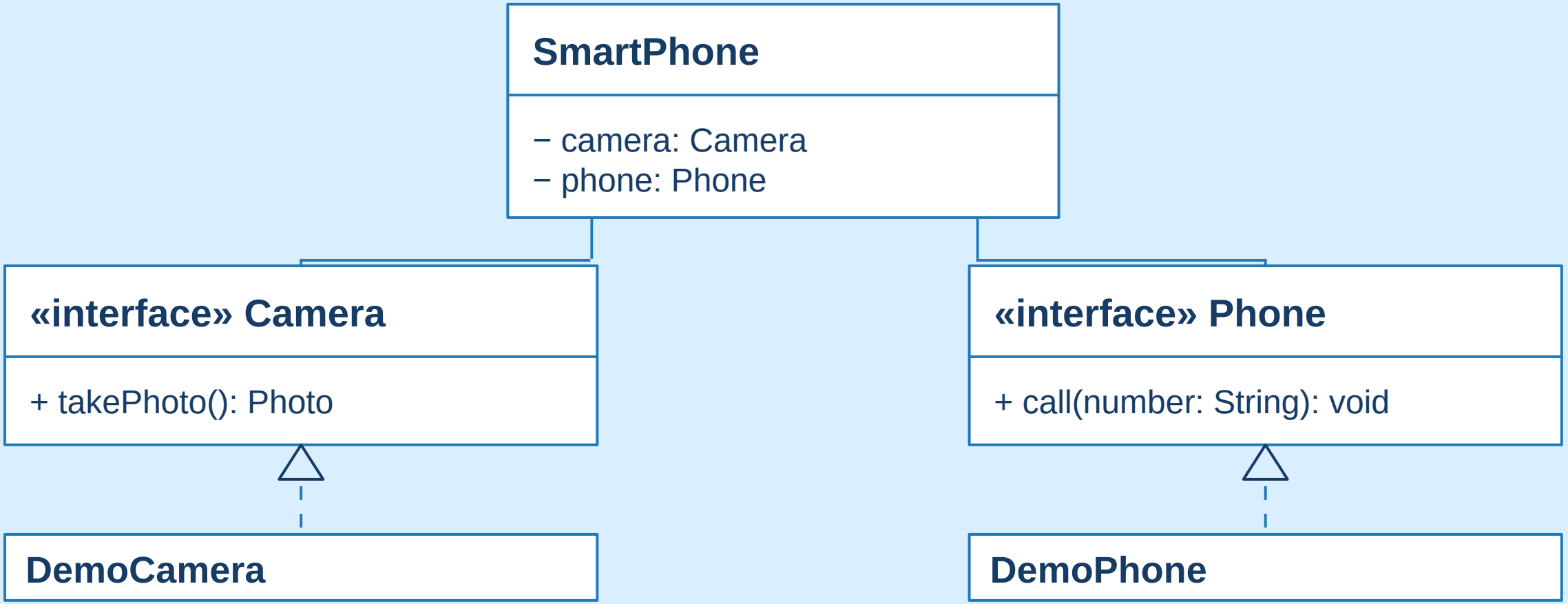
```
class SmartPhone {
  private final Camera camera;
  private final Phone phone;
  SmartPhone(Camera camera, Phone phone) {
    this.camera = java.util.Objects.requireNonNull(camera);
    this.phone = java.util.Objects.requireNonNull(phone);
  }
  public Photo takePhoto() { return camera.takePhoto(); }
  public void call(String number) { phone.call(number); }
}
```

Constructor supplies collaborators.

Methods delegate the work.

SmartPhone knows contracts rather than device details.

UML Structure of the Refactored Design



Solid associations show stored collaborators. Dashed realization links connect implementations to interfaces.

Assembling and Replacing Components

```
SmartPhone device = new SmartPhone(
    new DemoCamera(), new DemoPhone());
Photo photo = device.takePhoto();
device.call("12345");

Camera testCamera = () -> new Photo("Test image");
SmartPhone testDevice = new SmartPhone(
    testCamera, new DemoPhone());
```

Assembly selects concrete implementations.

Replacement preserves the Camera contract.

SmartPhone code stays the same.

Place these statements inside a method. Compatibility includes behaviour, not just matching signatures.

Cohesion, Coupling and Trade-Offs

Potential improvements

Feature implementations have clearer purposes.

Clients can reuse Camera or Phone separately.

Tests can substitute collaborators.

Costs and limits

The design adds interfaces and assembly work.

More links do not automatically mean worse coupling.

Poor contracts can still spread changes.

Testing Separated Responsibilities

Component checks

DemoPhone rejects null and blank input.
Camera returns the documented result.

Test each component against its contract.

Coordinator checks

Use a fake camera that records invocations.
Check that takePhoto() delegates once and returns its result.

Use a fake phone to inspect the supplied number.

Workshop: Speakers and Lighting

Current design

A Utility class contains speaker-volume controls and LED brightness controls.

Propose focused interfaces and implementations.

Required explanation

Which responsibilities belong together?
Who coordinates them?
Which component could be reused?
What contract makes replacement safe?

Possible Workshop Solution

Focused components

Speaker: setVolume(level)
Lighting: setBrightness(level)

Each implementation handles its own device details and validation.

Coordinator and contracts

A DeviceController delegates to both roles.

Define accepted ranges and error behaviour.

A mute operation belongs to audio, not lighting.

If a scene changes both sound and light, keep scene coordination separate from each device mechanism.

Review and Key Ideas

Separation of concerns

Identify meaningful responsibilities.
Group related state and behaviour.
Hide implementation behind contracts.
Delegate work to suitable collaborators.

Design review questions

Can each class’s purpose be explained?
What changes independently?
Where does coordination belong?
Can components be tested and replaced?

Choose boundaries that make the actual system easier to understand and change.