FARABI UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Object-Oriented Analysis and Design

Lecture 5
Creating Models in Design and Language Evolution

Dr. Bazargul Matkerim, Professor-Researcher

Learning Outcomes

Model	Translate	Compare
Distinguish domain concepts from implementation details.	Connect object roles, UML and Java code.	Explain paradigms and the evolution of abstraction.

Learning task: explain how a design model guides implementation.

Models Help Us Reason About Software



- Select details that answer a design question.
- Discuss structure and behaviour before building.
- Document decisions and review their consequences.

A model is a purposeful simplification. It is not the entire system.

From Conceptual to Technical Design

Conceptual model

Identify meaningful domain concepts and the rules connecting them.

Example: Member borrows BookCopy.

Technical model

Assign responsibilities, attributes, operations and interfaces.

Example: LoanService coordinates borrowing checks.

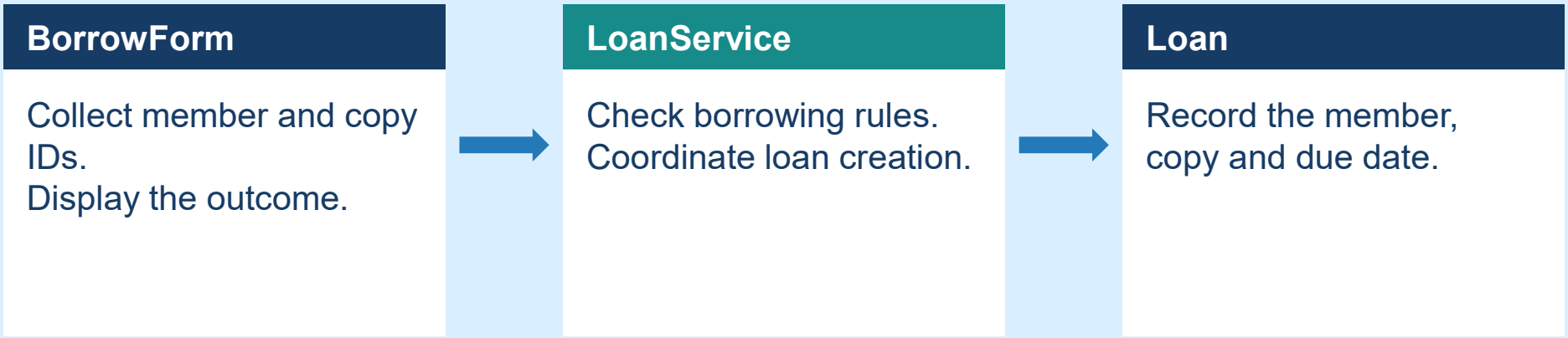
Refinement adds implementation detail while preserving the original user goal.

Entity, Boundary and Control Roles

Entity	Boundary	Control
Represents domain information and related behaviour. Example: Loan	Handles interaction with users or external systems. Example: BorrowForm	Coordinates a use-case flow across other objects. Example: LoanService

These are modelling roles, not mandatory categories for every software class.

Objects Collaborate Across Roles



The boundary delegates coordination; the entity represents domain state.

UML Offers Different Views

Structure	Behaviour
Class diagram Classes and relationships	Sequence diagram Messages over time
Component diagram Larger units and interfaces	State machine diagram States and transitions

Choose the UML view that answers the question. UML is a modelling language.

A Class Model Captures a Contract

BookCopy

− available: Boolean = true

+ isAvailable(): Boolean
+ borrow(): void

State: whether the copy can be borrowed.

Query: report current availability.

Command: borrow only if the copy is available.

Visibility: − private, + public. The operation contract includes its precondition.

From a Class Model to Java

```
class BookCopy {
  private boolean available = true;

  public boolean isAvailable() {
    return available;
  }

  public void borrow() {
    if (!available)
      throw new IllegalStateException();
    available = false;
  }
}
```

Private state

Public query

Guarded state change

Simplified Java example: additional loan rules and persistence are outside this class.

Models Need Continuous Evaluation

Check the model

Does it support the use case?
Are responsibilities clear?
Are failure outcomes represented?

Check the implementation

Does code preserve the contract?
Can tests expose invalid states?
Do model and code still agree?

Generated code skeletons can help, but they do not supply all behaviour or tests.

What Is a Programming Paradigm?

A paradigm is a style of organizing computation and expressing a solution.

Imperative	Declarative	Object-oriented
Express commands that update state.	Describe a desired result or a set of constraints.	Organize responsibilities around interacting objects.

These categories overlap. A language can support more than one paradigm.



Imperative Programming Makes Steps Explicit

```
int total = 0;
for (int value : values) {
  total += value;
}
```

- Initialize the accumulator.
- Visit each value in order.
- Update the accumulator at each step.

State can be local to a method or object. Imperative code need not use global variables.



Structured and Procedural Programming

Structured programming

Compose sequence, selection and repetition into clear control flow.

Examples: if, while, for

Procedural programming

Group related steps into procedures or functions with explicit inputs.

Example: calculateTotal(values)

Object methods can still use structured control flow and procedural decomposition.



Data Scope Limits Visibility

Shared global data

Many parts of a program can depend on the same mutable value.

Changes can have distant effects.

Local and private data

Local variables limit visibility to a scope. Private fields restrict direct access to object state.

Interfaces make access explicit.

Scope and encapsulation reduce exposure; good interfaces still require deliberate design.



Abstract Data Types Hide Representation

Stack: abstract contract

push(item)
pop()
isEmpty()

Last item in, first item out.

Possible representations

Array with a top index

Linked nodes

Clients use the same operations.

An abstract data type specifies values and operations without exposing representation.



Objects Combine State and Behaviour

Object A: BookCopy

Private state
available = true

Public operations
isAvailable(), borrow()

Object B: BookCopy

Private state
available = false

Public operations
isAvailable(), borrow()

Objects share a class definition while maintaining separate identities and state.

How Abstraction Has Evolved

Control abstraction

Procedures group steps. Structured control flow organizes execution.

Data abstraction

Modules and abstract data types hide representation behind operations.

Object abstraction

Objects combine state and behaviour; classes describe common structure.

These ideas developed along overlapping paths. New approaches did not replace all earlier ones.

Several Paradigms Can Coexist

Object-oriented

course.matches(keyword)

Ask an object to perform a responsibility.

Functional style

filter courses with a matching predicate

Compose transformations; favour pure functions.

Declarative query

```
SELECT title
FROM Course
WHERE level = 2
```

State the desired rows.

Examples illustrate different styles; the best choice depends on the task and context.

Practice: Model a Library Loan

Requirement: a member may borrow only an available book copy.

Build the model

Identify boundary, control and entity responsibilities.
Sketch a BookCopy class.
Write the borrow() contract.

Evaluate the design

Show a successful request.
Show a repeated request for the same copy.
Explain which object prevents it.

Connect every design decision to required behaviour rather than adding detail without purpose.

Review and Key Ideas

Models help teams understand, communicate and refine a design.
UML structure and behaviour views answer different questions.
Encapsulation connects data abstraction with object behaviour.
Programming paradigms overlap and can coexist in one solution.

**Exit question: why is a UML class diagram useful,
even when it cannot generate a complete working program?**