OOAD/UML 기본

건국대학교
유준범

Contents

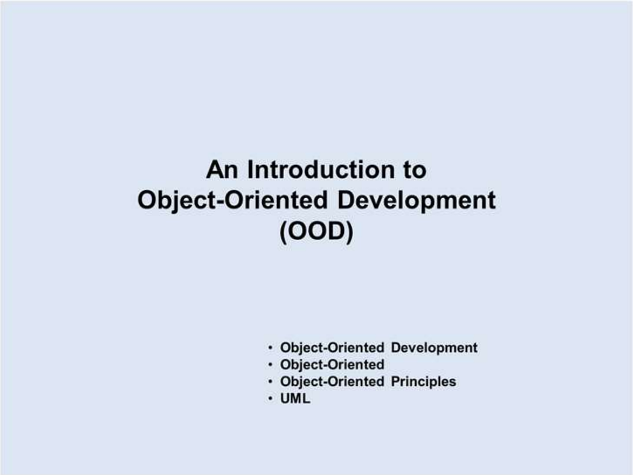
1. An Introduction to Object-Oriented Development
   - Object-Oriented Development
   - Object-Oriented
   - Object-Oriented Principles
   - UML

2. Object-Oriented Analysis and Design
   - Part I. Introduction
   - Part II. Inception
   - Part III. Elaboration Iteration 1 - Basics
     - OO Analysis
     - OO Design
     - OO Implementation

3. Advanced Topics in UML
   - Statechart Diagram
   - Component Diagram
   - Extension Mechanism of UML

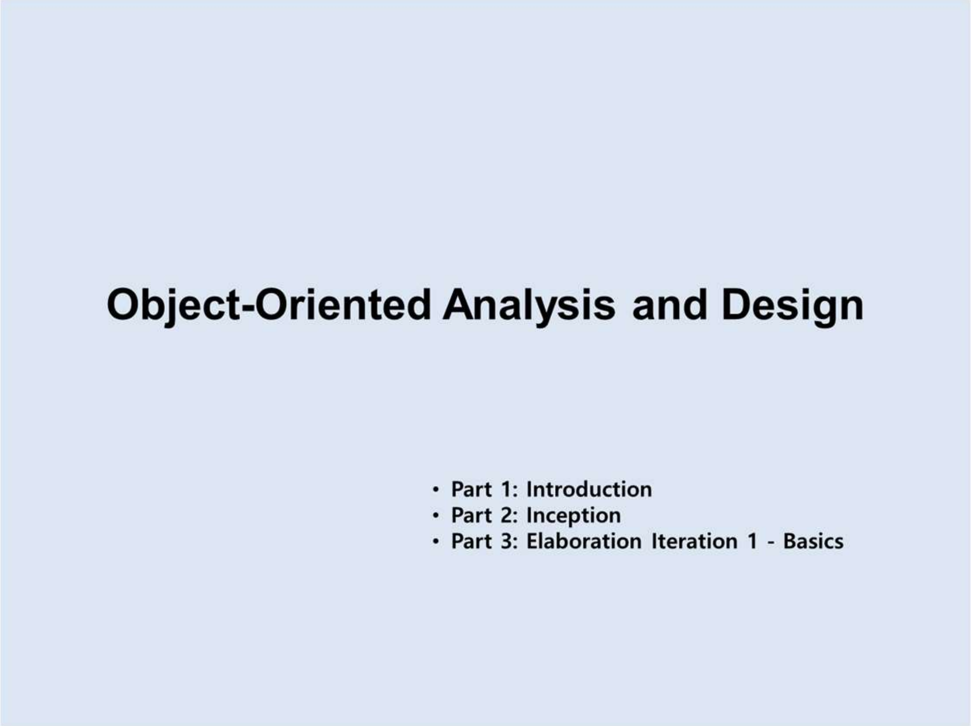
4. Object-Oriented Analysis and Design - Summary

Contents at a Glance



An Introduction to Object-Oriented Development (OOD)

- Object-Oriented Development
- Object-Oriented
- Object-Oriented Principles
- UML

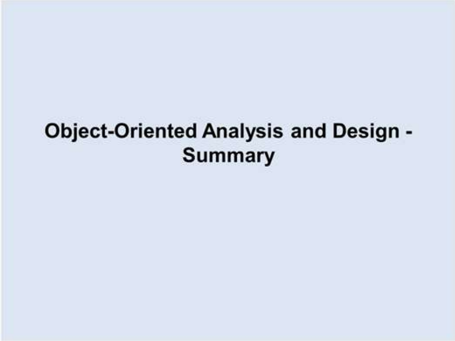


Object-Oriented Analysis and Design

- Part 1: Introduction
- Part 2: Inception
- Part 3: Elaboration Iteration 1 - Basics

Advanced Topics in UML

- Statechart Diagram
- Component Diagram
- Extension Mechanism of UML

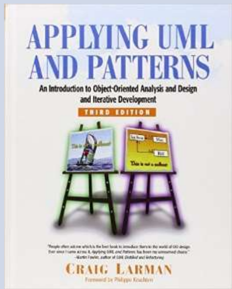


Object-Oriented Analysis and Design - Summary

Contents in Detail

대주제	차시	소주제	학습 목표	상세 내용
1. An Introduction to Object-Oriented Development	1 2	Object-Oriented Development	• '소프트웨어 개발'을 정의할 수 있다. • OOAD 와 SASD의 차이점을 구분할 수 있다. • 다양한 소프트웨어 개발 방법론/프로세스를 구분하고 정리할 수 있다.	• OOAD vs. SASD • Software Development Process
	3	Object-Oriented	• 객체지향 (Object-Oriented)을 정의할 수 있다.	• Object-Oriented
	4	Object-Oriented Principles	• 객체지향 Principles을 이해하고 적용할 수 있다.	• Object-Oriented Principles
	5 6	UML	• UML 2.0을 구성하는 13개 다이어그램들의 목적을 이해할 수 있다.	• 13 UML Diagrams

Contents in Detail

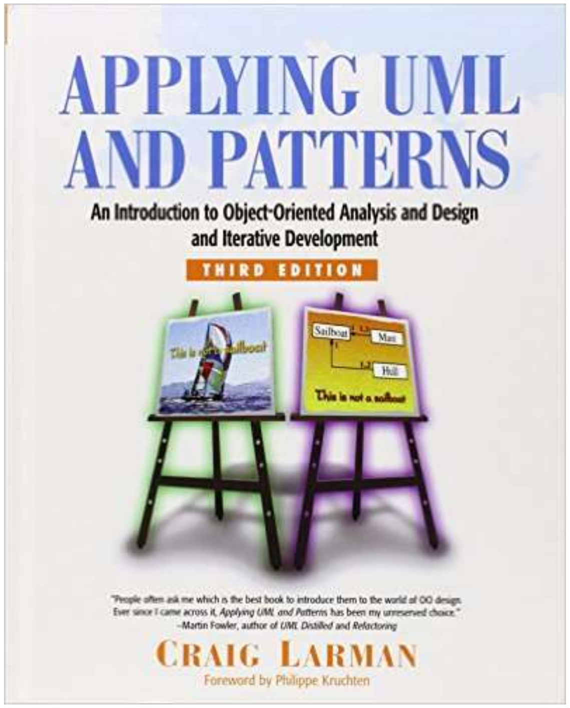
대주제	차시	소주제	학습 목표	상세 내용
2. Object-Oriented Analysis and Design  APPLYING UML AND PATTERNS An Introduction to Object-Oriented Analysis and Design and Iterative Development THIRD EDITION CRAIG LARMAN	7	Part I. Introduction	• OOAD 및 UP 기본개념을 정리할 수 있다. • 교재의 Case Study 내용을 확인할 수 있다.	• Chapter 1. Object-Oriented Analysis and Design • Chapter 2. Iterative, Evolutionary, and Agile • Chapter 3. Case Studies
	8 9	Part II. Inception	• UP 기반 OOAD의 첫 단계인 Inception 단계를 이해할 수 있다. • Inception 단계의 활동을 수행할 수 있다. • 기능/비기능 요구사항을 구별할 수 있다. • Use Case를 활용할 수 있다.	• Chapter 4. Inception is Not the Requirements Phase • Chapter 5. Evolutionary Requirements • Chapter 6. Use Cases • Chapter 7. Other Requirements
	10	Part III. Elaboration Iteration 1 – Basics - OOA	• Analysis 단계의 활동을 이해할 수 있다. • Domain model의 목적을 이해하고 활용할 수 있다.	• Chapter 8. Iteration 1 Basics • Chapter 9. Domain Models
	11	- OOA	• Sequence diagram의 목적을 이해하고 활용할 수 있다. • Operation contract의 목적을 이해할 수 있다.	• Chapter 10. System Sequence Diagram • Chapter 11. Operation Contracts
	12	- OOD	• Design 단계의 활동을 이해할 수 있다. • Package diagram의 목적을 이해하고 활용할 수 있다.	• Chapter 12. Requirements to Design Iteratively • Chapter 13. Logical Architecture and UML Package Diagrams
	13 14	- OOD	• Sequence diagram의 목적을 이해하고 활용할 수 있다.	• Chapter 14. On to Object Design • Chapter 15. UML Interaction Diagram
	15 16	- OOD	• Class diagram의 목적을 이해하고 활용할 수 있다.	• Chapter 16. UML Class Diagram
	17	- OOD	• GRASP 디자인 패턴의 목적과 효과적인 적용 방법을 이해할 수 있다.	• Chapter 17. GRASP: Designing Objects with Responsibilities
	18 19	- OOI	• OO Design에서 Implementation으로의 전환과정을 정확하게 이해할 수 있다. • 개발방법론의 장점을 확인할 수 있다.	• Chapter 19. Designing for Visibility • Chapter 20. Mapping Designs to Code

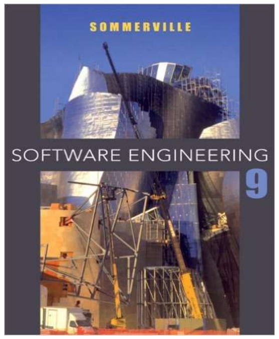
Contents in Detail

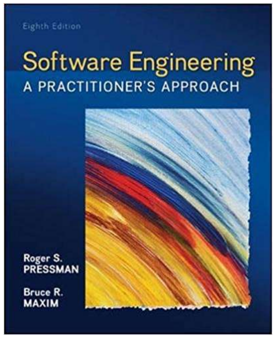
대주제	차시	소주제	학습 목표	상세 내용
3. Advanced Topics in UML	20 21	Statechart Diagram	• Statechart의 문법을 정확하게 이해하고, 이를 활용하여 모델링을 수행할 수 있다.	• Statechart Diagram
	22	Component Diagram	• Component Diagram을 이해하고 활용할 수 있다.	• Component Diagram
	23	Extension Mechanism of UML	• UML을 적절하게 확장하는 방법을 이해할 수 있다. • MOF의 개념을 이해할 수 있다.	• Extension Mechanism of UML

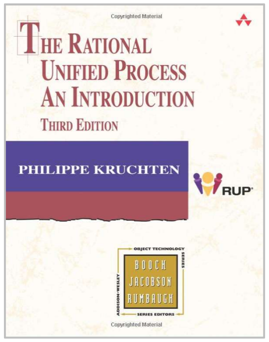
대주제	차시	소주제	학습 목표	상세 내용
4. Summary	24	OOAD Summary	• UML을 적절하게 사용하여, UP 기반의 OOAD를 수행할 수 있는 이론적인 배경을 갖춘다.	• OOAD Summary

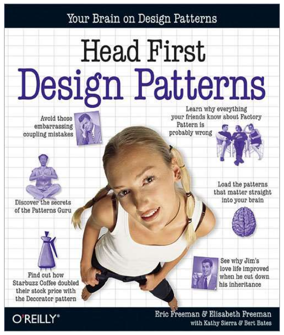
Text and References

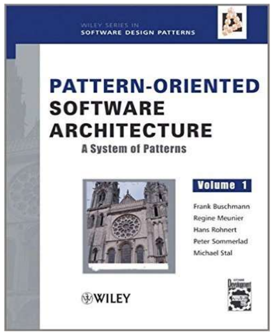

APPLYING UML AND PATTERNS
An Introduction to Object-Oriented Analysis and Design and Iterative Development
THIRD EDITION
CRAIG LARMAN
Foreword by Philippe Kruchten

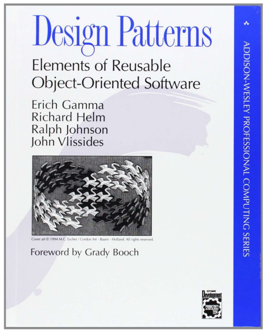







Your Brain on Design Patterns
Head First Design Patterns
O'REILLY
Eric Freeman & Elisabeth Freeman
with Kathy Sierra & Bert Bates




Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

An Introduction to Object-Oriented Development (OOD)

- Object-Oriented Development
- Object-Oriented
- Object-Oriented Principles
- UML

Object-Oriented Development

Software Development

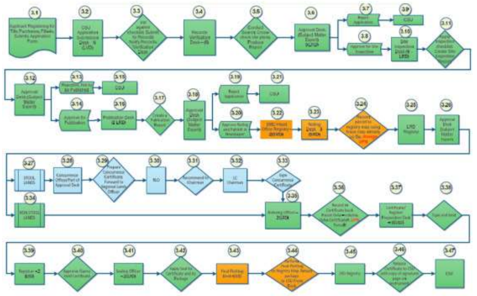
- Software Development ≈ Solving Problem with Software in Computer

Problems in real world

Natural Language
→ Descriptions of Problems
(through Identifying Requirements)

+

Business Process



A Big Gap between Languages

Solutions in computer

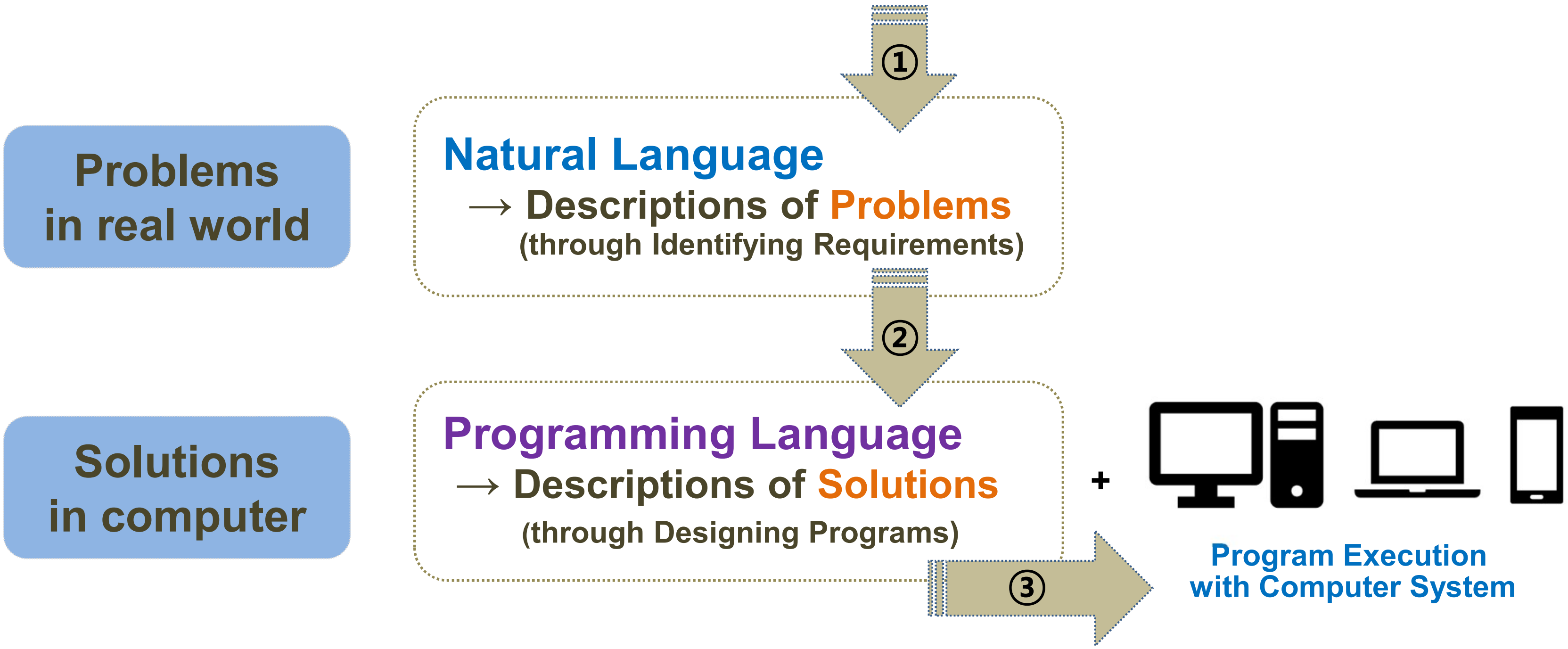
Programming Language
→ Descriptions of Solutions
(through Designing Programs)

+

Program Execution
with Computer System

Software Development

- Software Development ≈ Solving Problem with Software in Computer



Problems in real world

Natural Language
→ Descriptions of Problems
(through Identifying Requirements)

Solutions in computer

Programming Language
→ Descriptions of Solutions
(through Designing Programs)

+

Program Execution
with Computer System

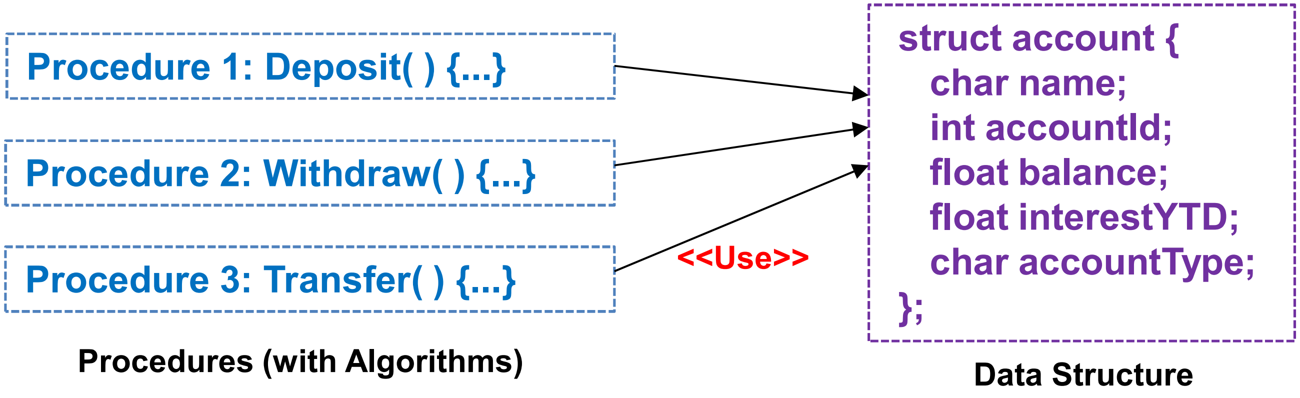
Software Development ≈ ①②③ → **Procedural Programming** → **SASD**

Software Development ≈ ①②③ → **Object-Oriented Programming** → **OOAD**

Computational Thinking ≈ ①②$_{/2}$

Procedural Programming

- A program is organized with **procedures**.
  - **Procedure/Function**
    - building-block of procedural programs
    - **statements** changing values of **variables**
  - Focusing on data structures, algorithms, and sequencing of steps
    - **Algorithm** : a set of instructions for solving a problem
    - **Data structure** : a construct used to organize data in a specific way
  - Most computer languages (from FORTRAN to C) are procedural ones.



Procedure 1: Deposit( ) {…}

Procedure 2: Withdraw( ) {…}

Procedure 3: Transfer( ) {…}

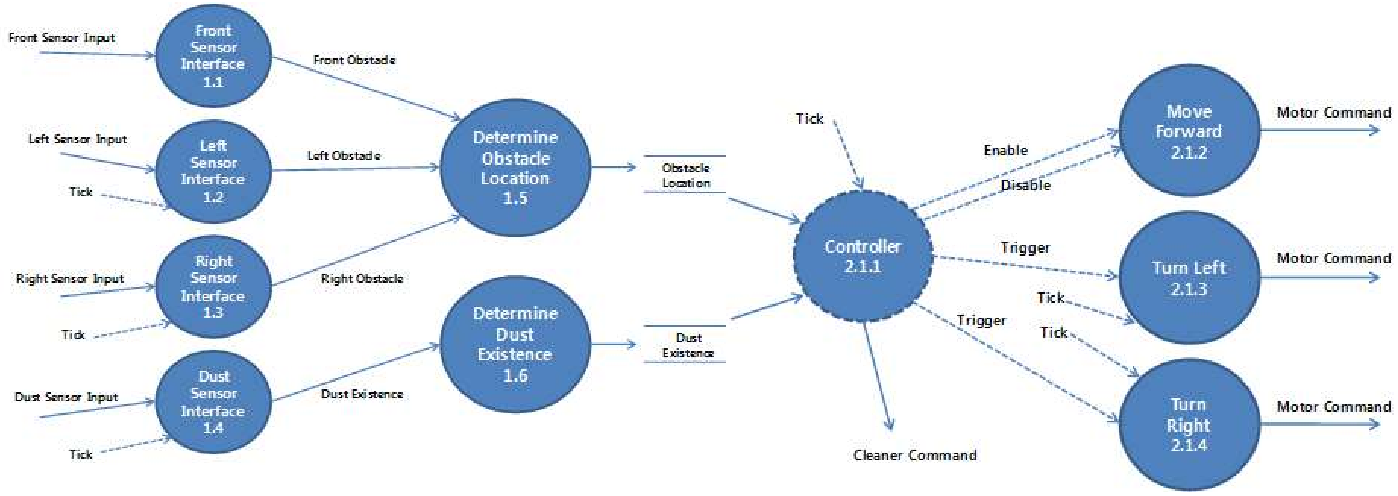
```
struct account {
   char name;
   int accountId;
   float balance;
   float interestYTD;
   char accountType;
};
```

Procedures (with Algorithms)

Data Structure

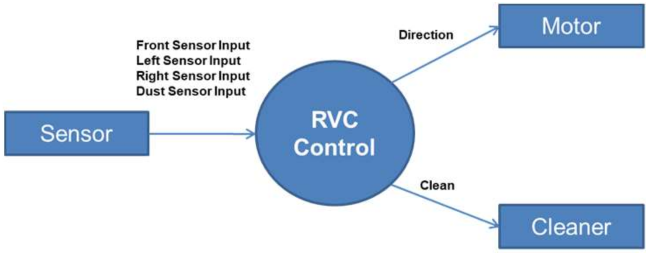
Procedural Programming - SASD

- **SASD** (Structured Analysis and Structured Design)
  - A traditional software development methodology for procedural programs
  - Top-Down Divide and Conquer
    - Divide large, complex problems into smaller, more easily handled ones.
  - Functional view of the problem using **DFD** (Data Flow Diagram)



A level 3 DFD for RVC Control

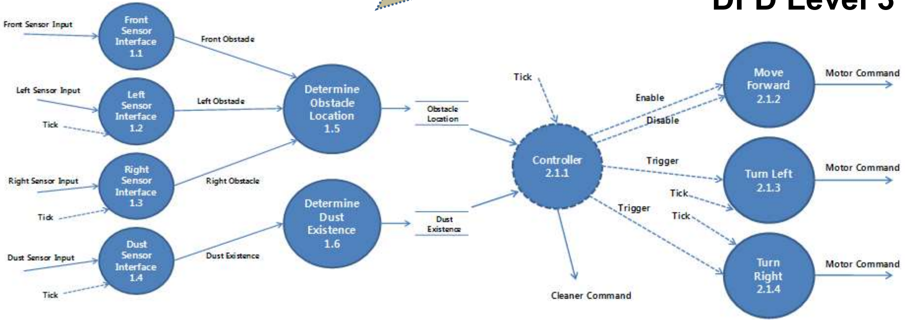
An SASD Example - RVC Control



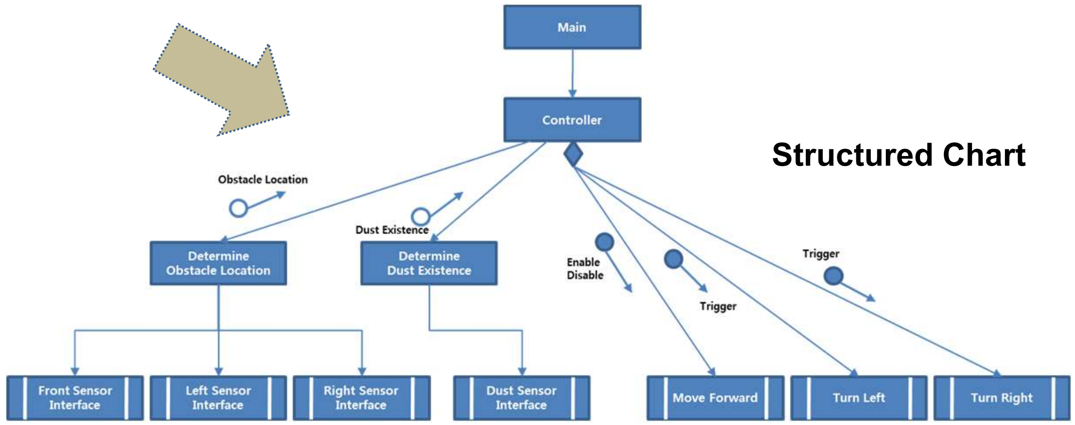
DFD Level 0

…

DFD Level 3



Structured Analysis



Structured Chart

Structured Design

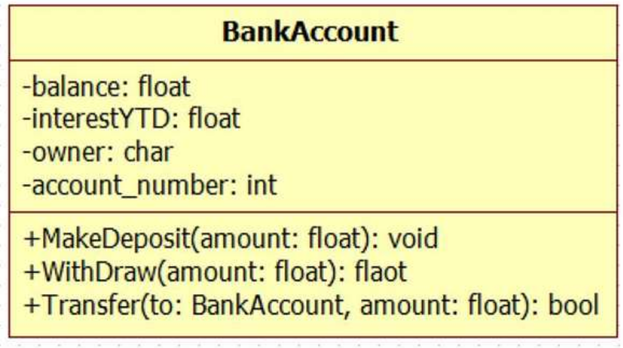
(…)

Object-Oriented Programming

- A program is organized with **objects**.
  - Focusing on objects and their communications.
    - **Object** : consisting of **data** and **operations** (functions)
    - **Object communication** : an object **calls** an operation of other objects with its data
  - Providing system functionalities through object communications
    - No explicit data flow
    - Only **communication sequences** among objects

data

operation



BankAccount

-balance: float
-interestYTD: float
-owner: char
-account_number: int

+MakeDeposit(amount: float): void
+WithDraw(amount: float): flaot
+Transfer(to: BankAccount, amount: float): bool

```
Class BankAccount {
  private:
    float balance;
    float interestYTD;
    char * owner;
    int account_number;
  public:
    void Deposit (float amount) {…}
    float WithDraw (float amount) {…}
    bool Transfer (BankAccount to, float amount) {…}
};
```

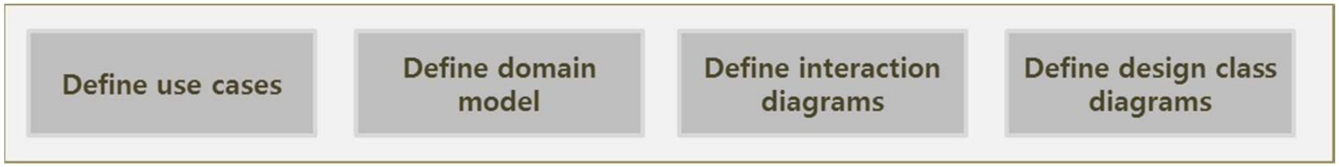
Object-Oriented Programming - OOAD

- **OOAD** (Object-Oriented Analysis and Design)
  - A software development methodology for Object-Oriented programs
  - OOA + OOD

- **Object-Oriented Analysis (OOA)**
  - Discover the domain concepts/objects (the objects of the problem domain)

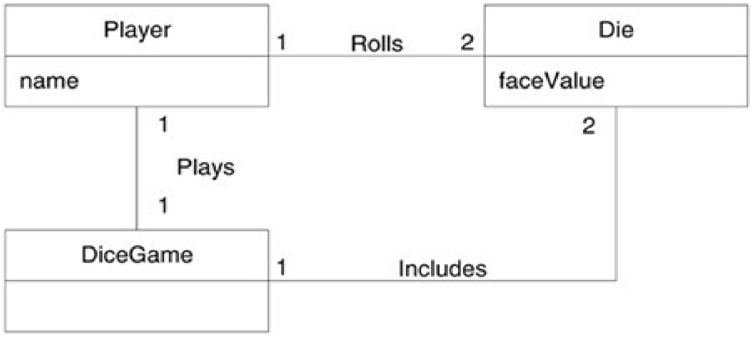
- **Object-Oriented Design (OOD)**
  - Define software objects (static)
  - Define how they collaborate to fulfill the requirements (dynamic)

An OOAD Example - Dice Game



OOA

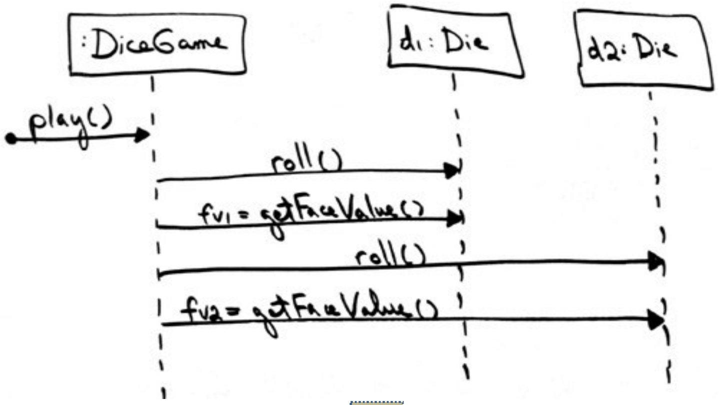
Use Case : Play a Dice Game
- **Player requests to roll the dice.**
- **System presents results.**
- **If the dice's face value totals seven, player wins; otherwise, player loses.**

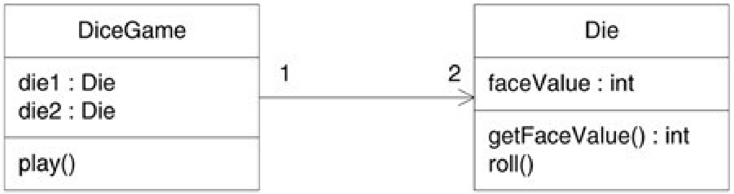


Domain Model

OOD

Interaction Diagram

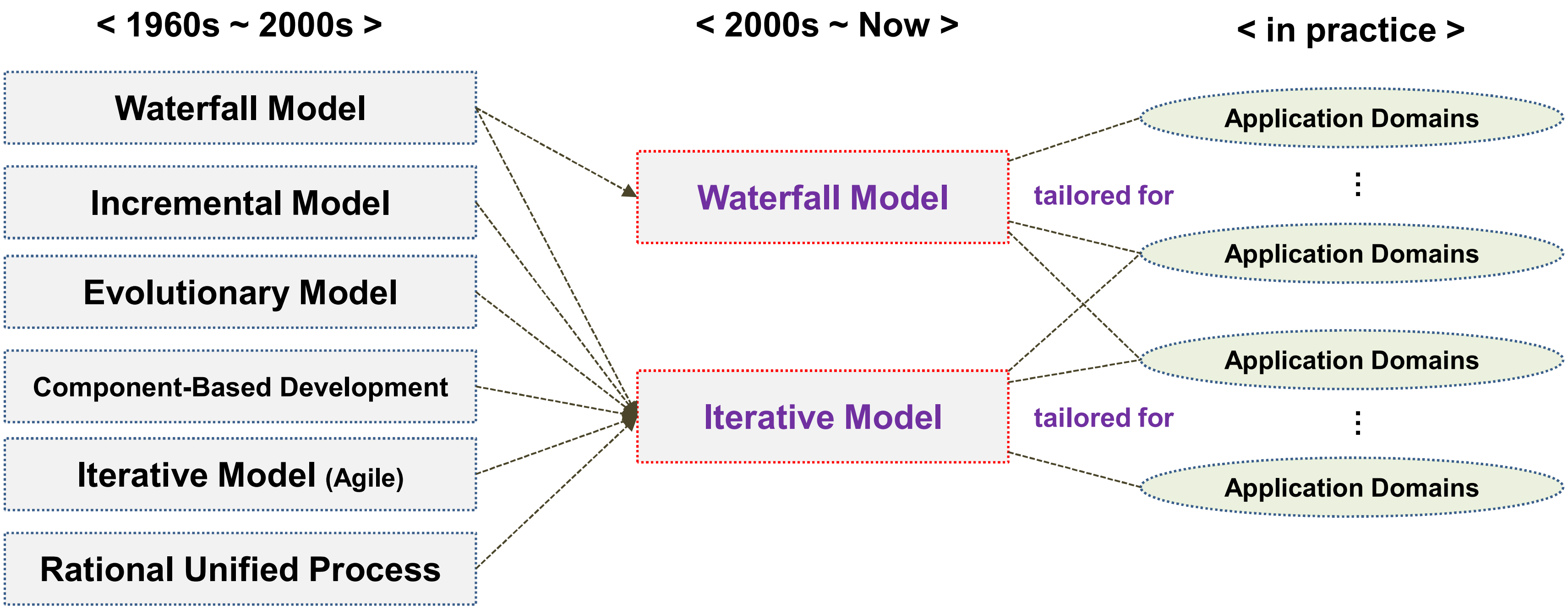




Design Class Diagram

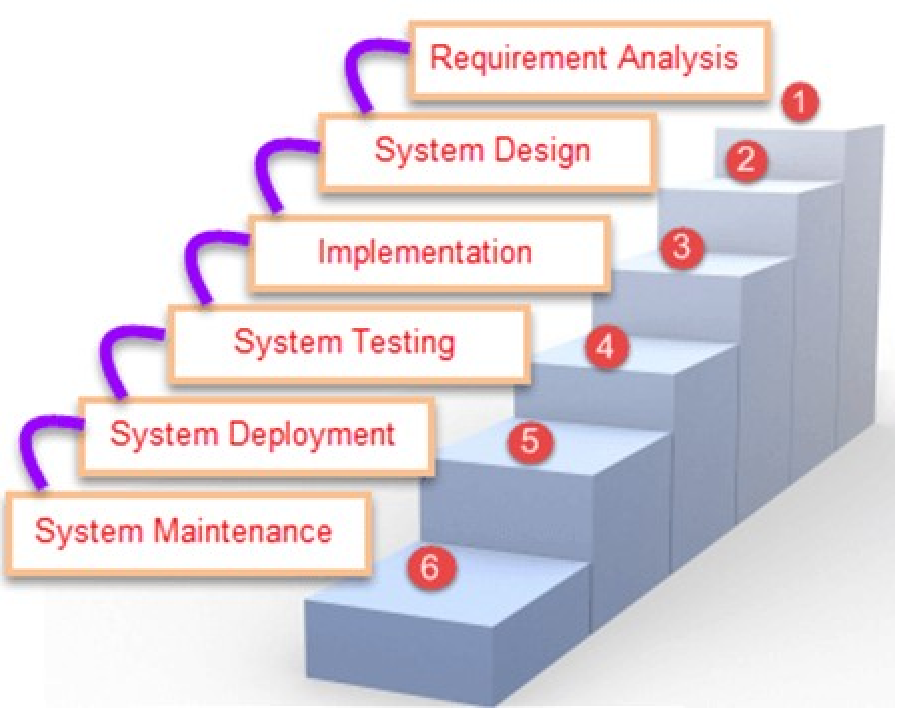
Software Process Model

- **Software (Development) Process models**
  - Defining a distinct set of activities, actions, tasks, milestones, and work products that are required to engineer high-quality software, systematically.
  - Defining **Who** is doing **What**, **When** to do it, **How** to reach a certain goal.



Waterfall Model

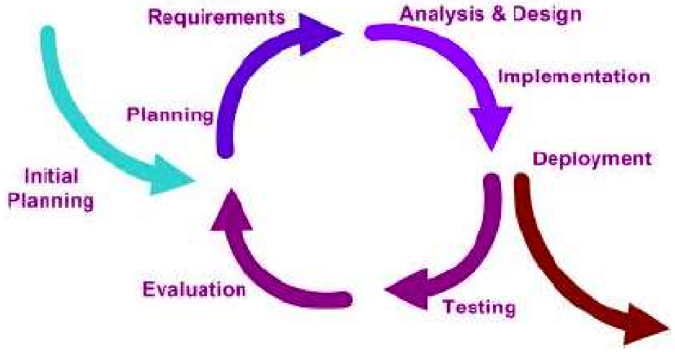
- A classic software development life-cycle (SDLC) model
  - Suggests a systematic and sequential approach to software development
  - Useful in situations where,
    - Requirements are fixed early.
    - Work can/shoudl proceed to completion in a linear manner.



Iterative Model - Agile

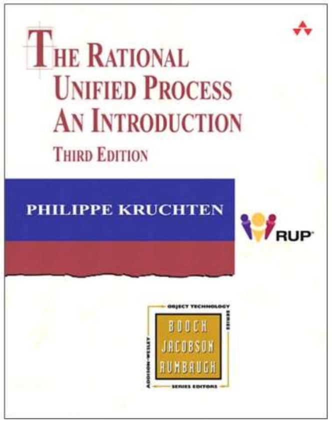
- **Agile development** is **an umbrella term** a group of methodologies weighting rapid prototyping and rapid development experiences.
  - Lightweight in terms of documentation and process specification
  - Example: XP(eXtreme Programming) , TDD(Test Driven Development)

- Agile methods attributes
  - Iterative (several cycles)
  - Incremental (not delivering the product at once)
  - Actively involve users to establish requirements

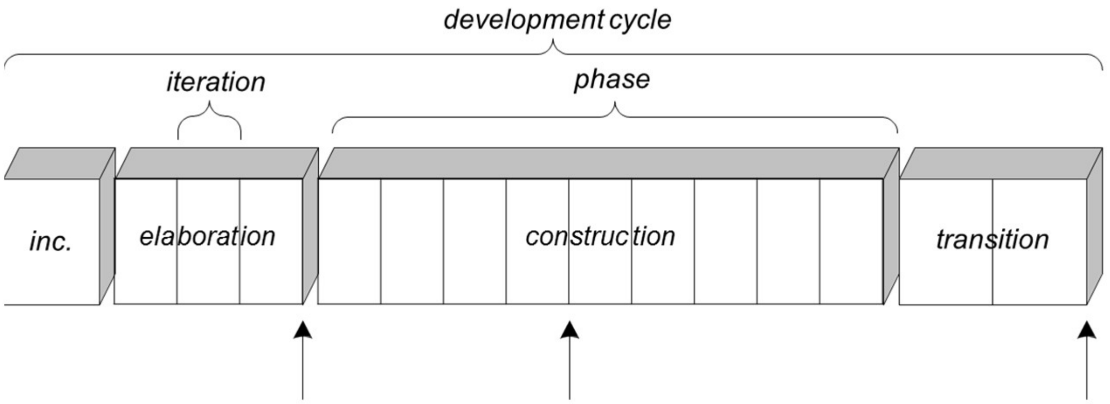


- Agile Manifesto
  - Individual over processes and tools
  - Working software over documentation
  - Customer collaboration over contract negotiation
  - Responding to change over following a plan

Iterative Model - UP



- **Rational Unified Process (RUP) or UP**
  - A Software development approach that is
    - **Iterative (Incremental, Evolutionary)**
      - Each iteration includes a small waterfall cycle.
    - **Risk-driven / Client-driven / Architecture-centric**
    - **Use-case-driven**
  - A Well-defined and well-structured software engineering process
    - 4 Phases and 9 Disciplines
  - A de-facto industry standard for developing OO software



development cycle

iteration

phase

inc.

elaboration

construction

transition

milestone

An iteration end-point when some significant decision or evaluation occurs.

release

A stable executable subset of the final product. The end of each iteration is a minor release.

increment

The difference (delta) between the releases of 2 subsequent iterations.

final production release

At this point, the system is released for production use.

3

An Introduction to Object-Oriented

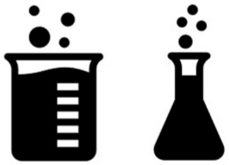
Object

- An **object** represents an **entity**.
  - physical, conceptual or software, informally.

  - Physical entity

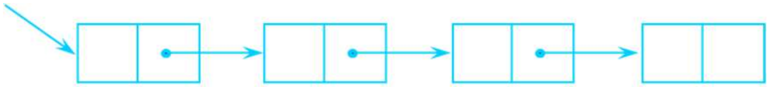
Truck

  - Conceptual entity



Chemical Process

  - Software entity



Linked List

A More Formal Definition of Object

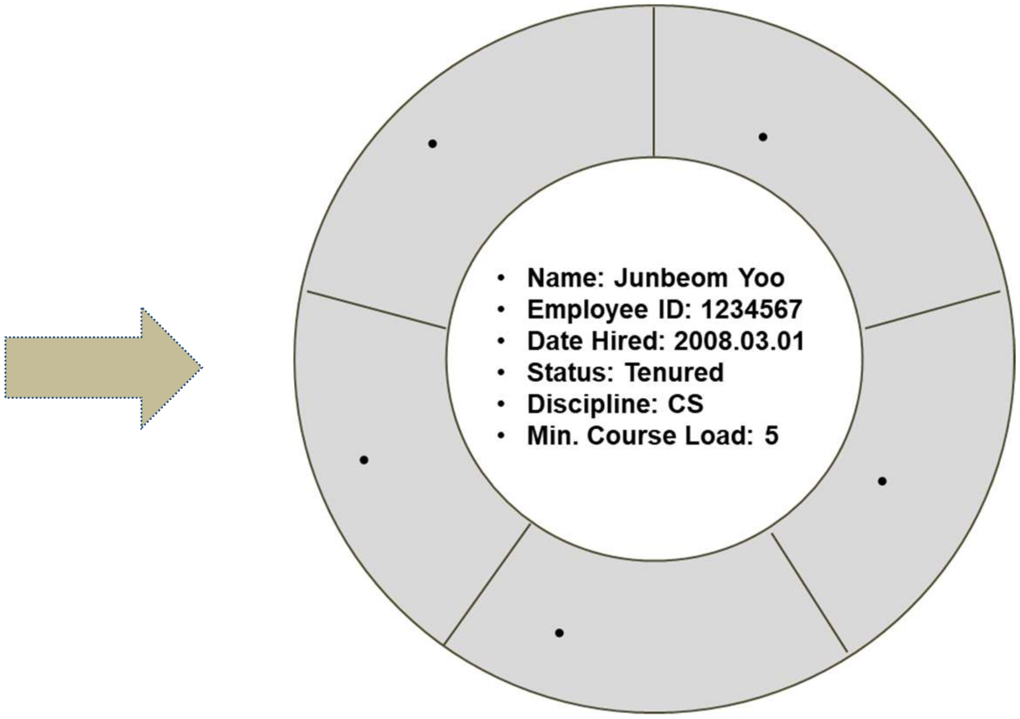
- An **object** is an **entity** with a well-defined boundary and identity that encapsulates **state** and **behavior**.
  - **State** : represented by attributes and relationships
  - **Behavior** : represented by operations, methods, and state machines

The Object States

- The **state** of an object
  - One of the possible conditions in which an object may exist.
  - Normally changes over time.

Professor Yoo

Name: Junbeom Yoo
Employee ID: 1234567
Date Hired: 2008.03.01
Status: Tenured
Discipline: CS
Min. Course Load: 5 classes



- Name: Junbeom Yoo
- Employee ID: 1234567
- Date Hired: 2008.03.01
- Status: Tenured
- Discipline: CS
- Min. Course Load: 5

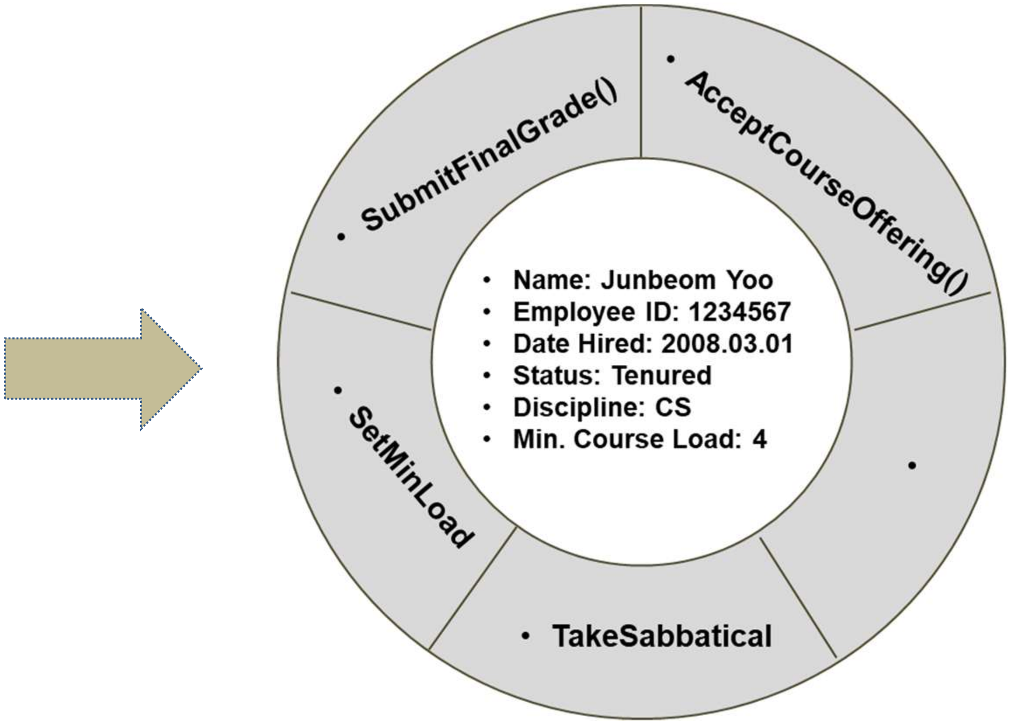
Professor Yoo

The Object Behavior

- **Behavior** determines how an object acts and reacts.
  - Modeled by the set of **messages** it can respond to (= operations the object can perform).

Professor Yoo

Name: Junbeom Yoo
Employee ID: 1234567
Date Hired: 2008.03.01
Status: Tenured
Discipline: CS
Min. Course Load: 5 classes

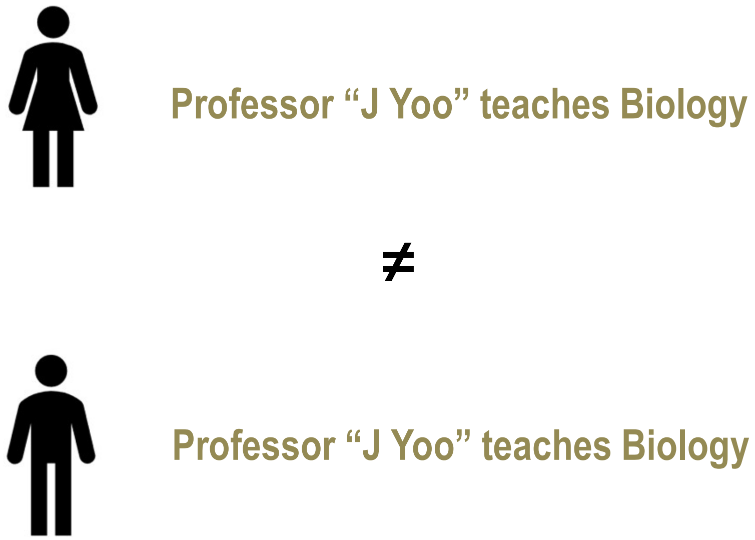


- Name: Junbeom Yoo
- Employee ID: 1234567
- Date Hired: 2008.03.01
- Status: Tenured
- Discipline: CS
- Min. Course Load: 4

Professor Yoo

An Object has Identity

- Each object has a **unique identity**.
  - Even if the state is identical to that of another object.



Professor “J Yoo” teaches Biology

≠

Professor “J Yoo” teaches Biology

Objects Need to Collaborate

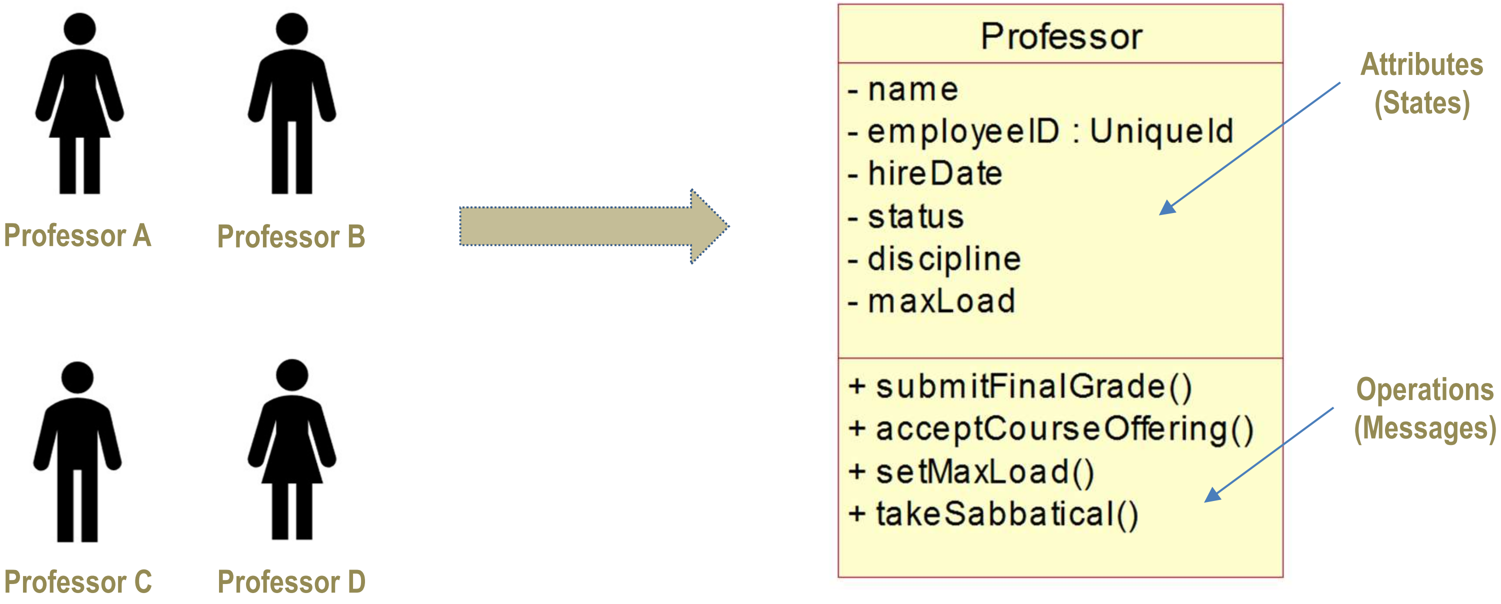
- Objects are useful only when they **can collaborate together** to solve a problem.
  - Each object is responsible for its own behavior and status.
  - No one object can carry out every responsibility on its own.

- How do objects interact with each other?
  - They interact through **messages**.

Class

- A **class** is a description of a set of objects that share the same properties and behavior.
  - **An object is an instance of a class.**

Objects | **Class**



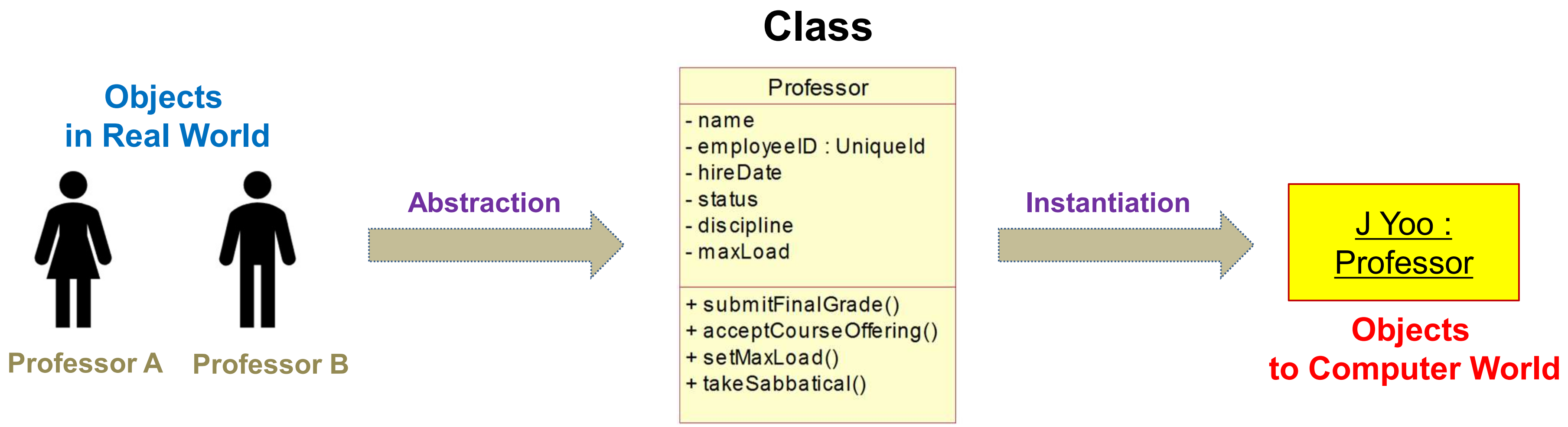
Professor

- name
- employeeID : UniqueId
- hireDate
- status
- discipline
- maxLoad

+ submitFinalGrade()
+ acceptCourseOffering()
+ setMaxLoad()
+ takeSabbatical()

Relationship between Classes and Objects

- **A class is an abstract definition of an object.**
  - It defines the structure and behavior of each object in the class.
  - It serves as a template for creating objects.
    - Objects are grouped into classes.
    - An object is an instance of a class.



Class

Objects in Real World

Professor A

Professor B

Abstraction

Professor
- name
- employeeID : UniqueId
- hireDate
- status
- discipline
- maxLoad

+ submitFinalGrade()
+ acceptCourseOffering()
+ setMaxLoad()
+ takeSabbatical()

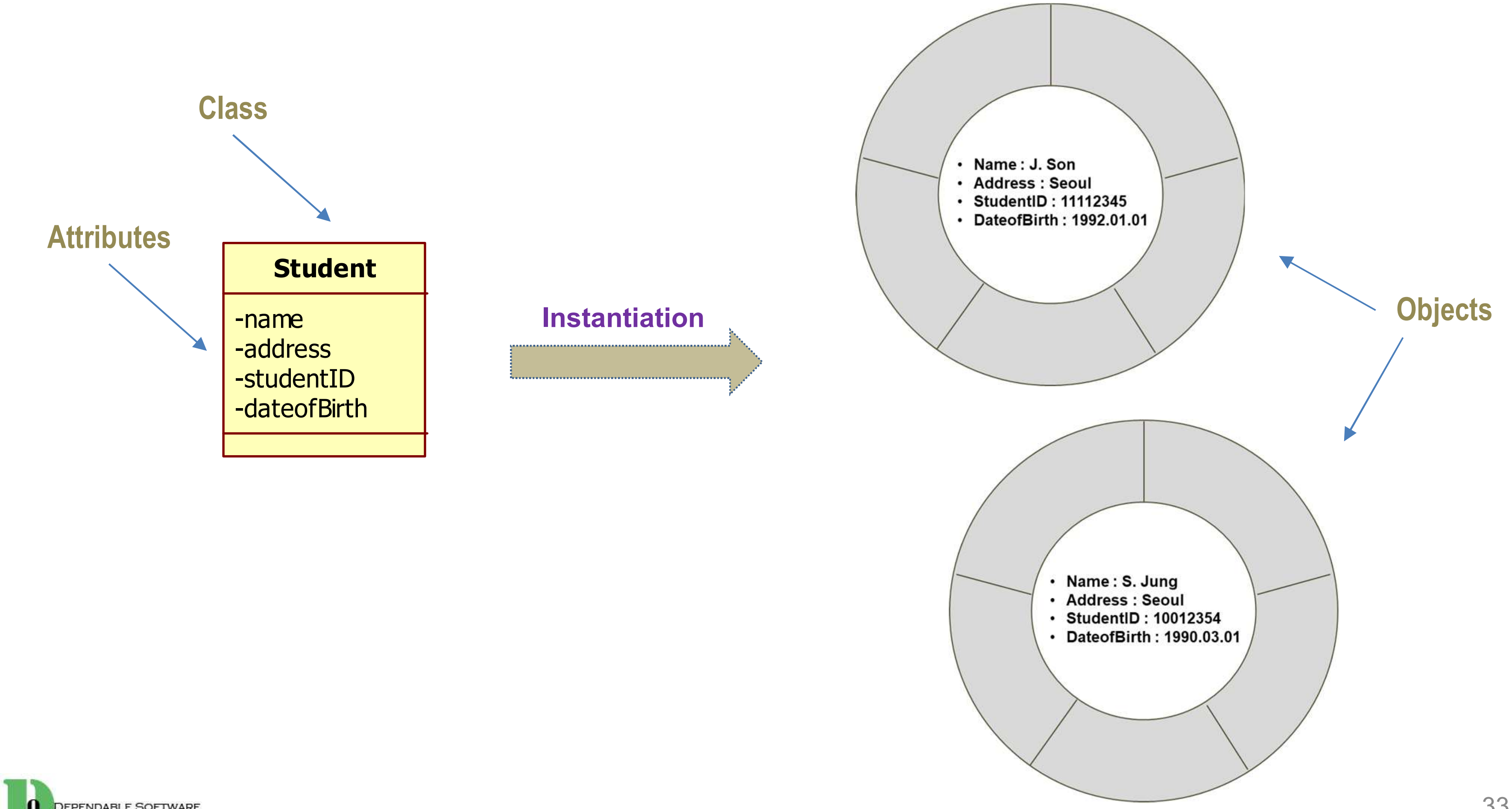
Instantiation

J Yoo : Professor

Objects to Computer World

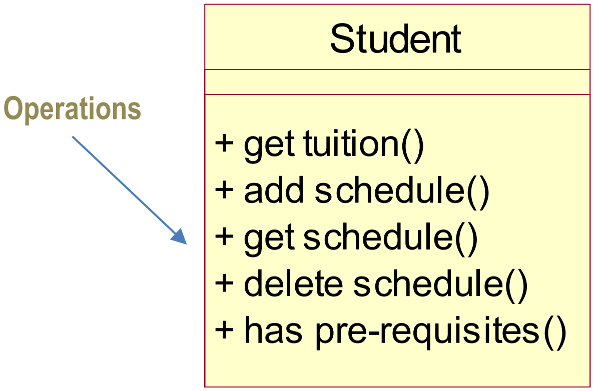
Attribute

- An **attribute** is **a named property of a class** that describes a range of values which instances of the property may hold.



Operation

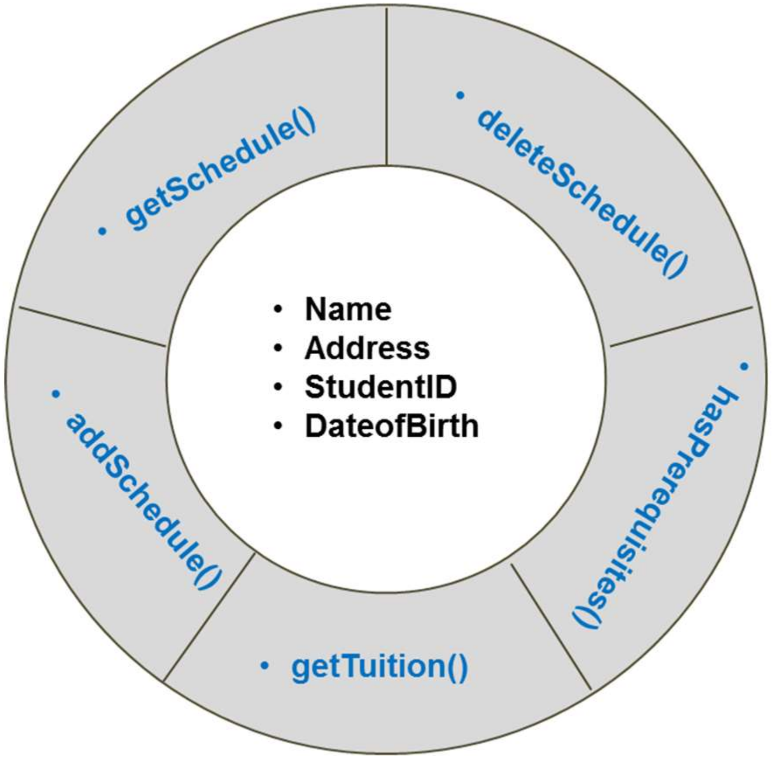
- An **operation** is <u>**the implementation of a service**</u> which can be requested from any object of the class to affect behavior.



Operations

Student

+ get tuition()
+ add schedule()
+ get schedule()
+ delete schedule()
+ has pre-requisites()



- getSchedule()
- deleteSchedule()
- hasPrerequisites()
- getTuition()
- addSchedule()

- Name
- Address
- StudentID
- DateofBirth

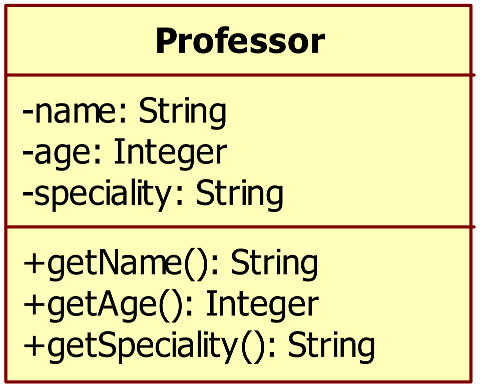
Example : class Professor

```
class Professor {
   private String name;
   private int age;
   private String specialty;

   public Professor (String sm, int ia, String ss) {
           name = sm;
           age = ia;
           speciality = sst;
   }

   public String getName ( ) { return  name;}
   public int getAge ( ) { return  age;}
   public String getSpeciality ( ) { return specialty;}
}
```

class



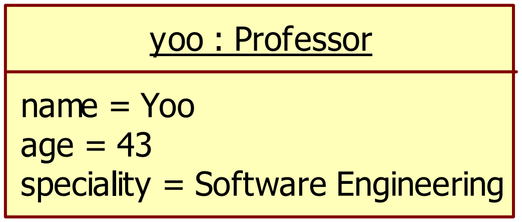
Professor

-name: String
-age: Integer
-speciality: String

+getName(): String
+getAge(): Integer
+getSpeciality(): String

```
Professor yoo = new Professor ( “yoo” , 43,  “Software Engineering” );
```

instance

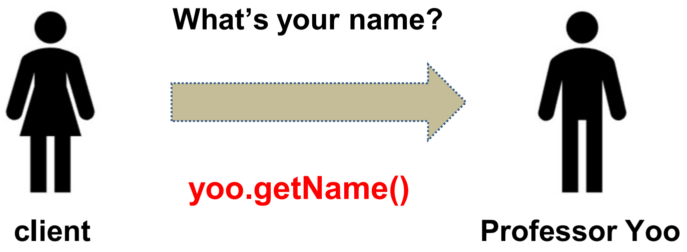


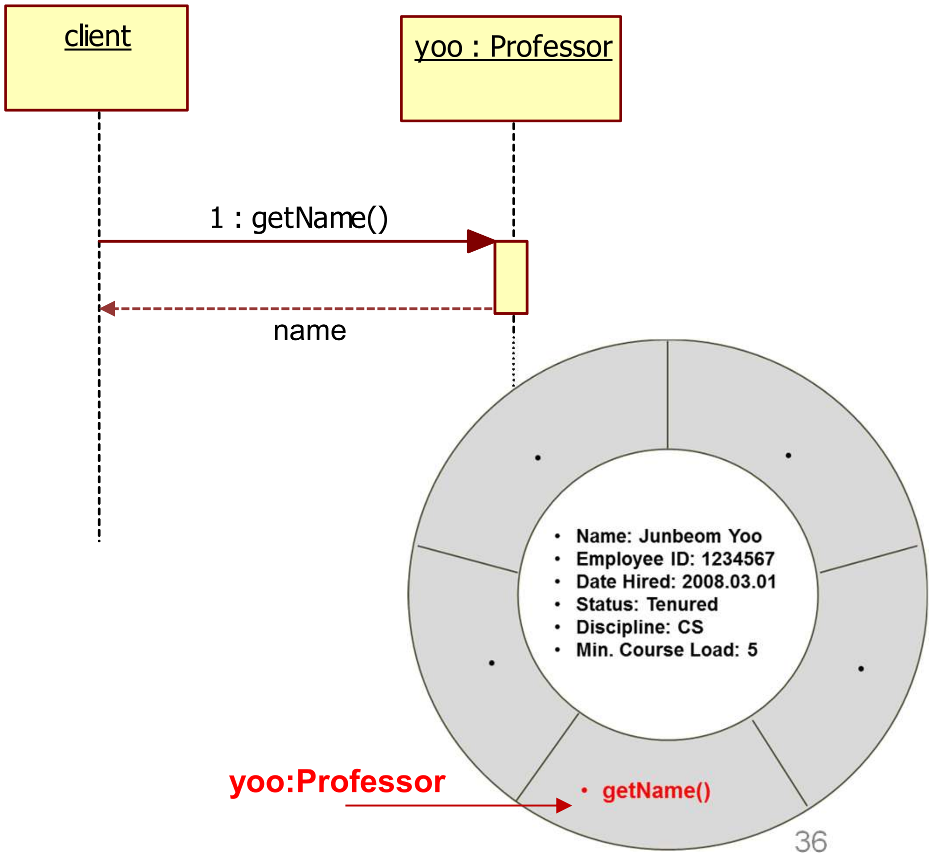
yoo : Professor

name = Yoo
age = 43
speciality = Software Engineering

Message

- **A specification of a communication** between objects
  - Conveying information with the expectation that activity will ensue.
  - One object asks another object to perform an operation.





4

An Introduction to Object-Oriented Principles

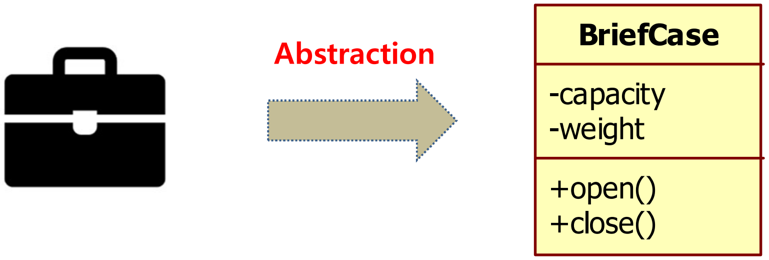
Basic Principles of Object-Oriented

1. Abstraction
2. Encapsulation
3. Inheritance
4. Polymorphism
5. Composition
6. Abstract / Interface Class

1. Abstraction

- **Abstraction** :
  - *"Any model that includes the most important, essential or distinguishing aspects of something while suppressing or ignoring less important, immaterial, or diversionary details. The result of removing distinctions so as to emphasize commonalties."*

    (Dictionary of Object Technology, Firesmith, Eykholt, 1995)
  - Emphasizes relevant characteristics, but suppresses other characteristics



BriefCase

-capacity
-weight

+open()
+close()

Example : Abstraction

Lecturer

Dormitory

Course Offering

Course (Chemistry)

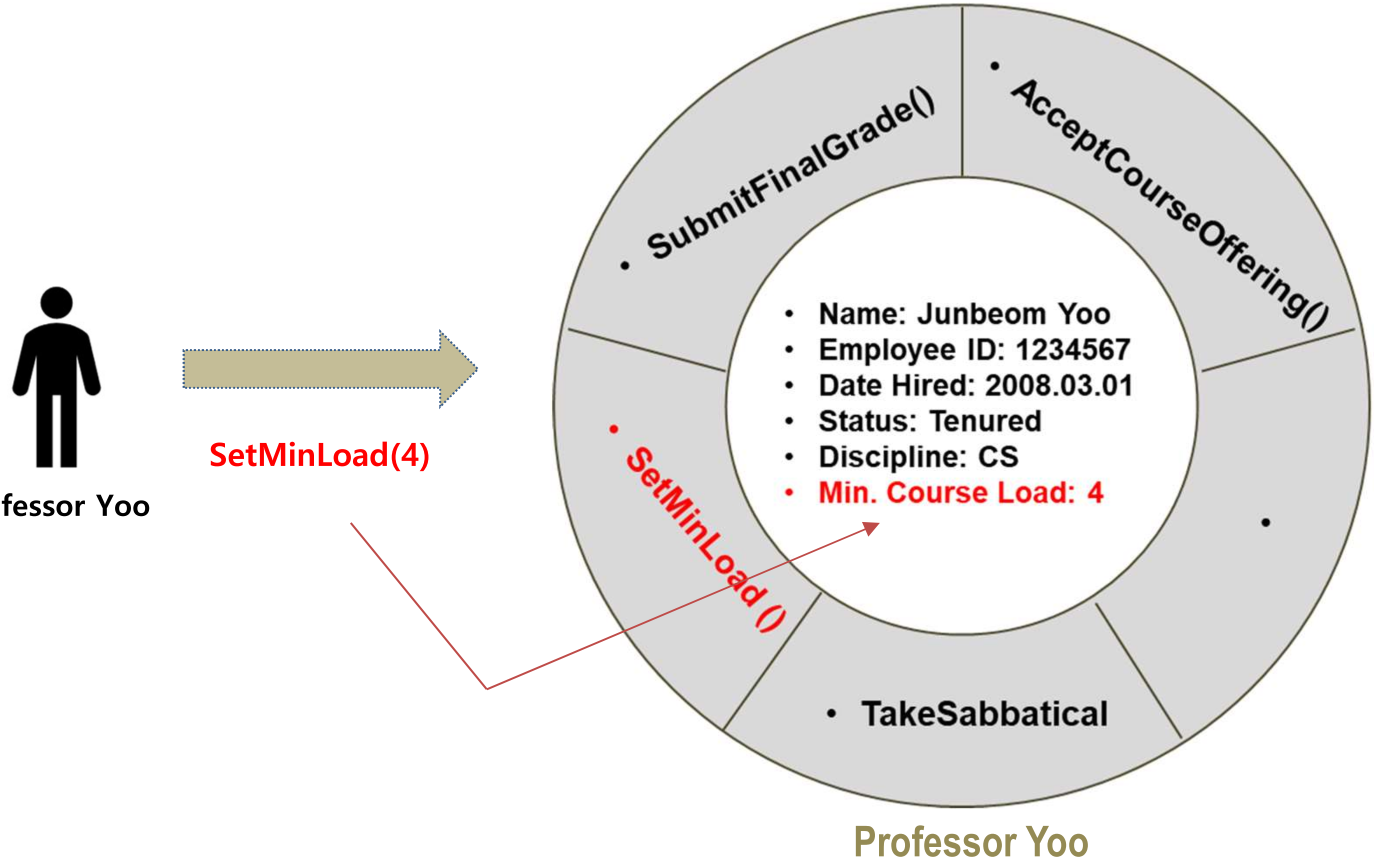
2. Encapsulation

- **Encapsulation** :
  - Design, produce and describe software so that it can be easily used without knowing the details of how it works.
  - Also known as **information hiding**

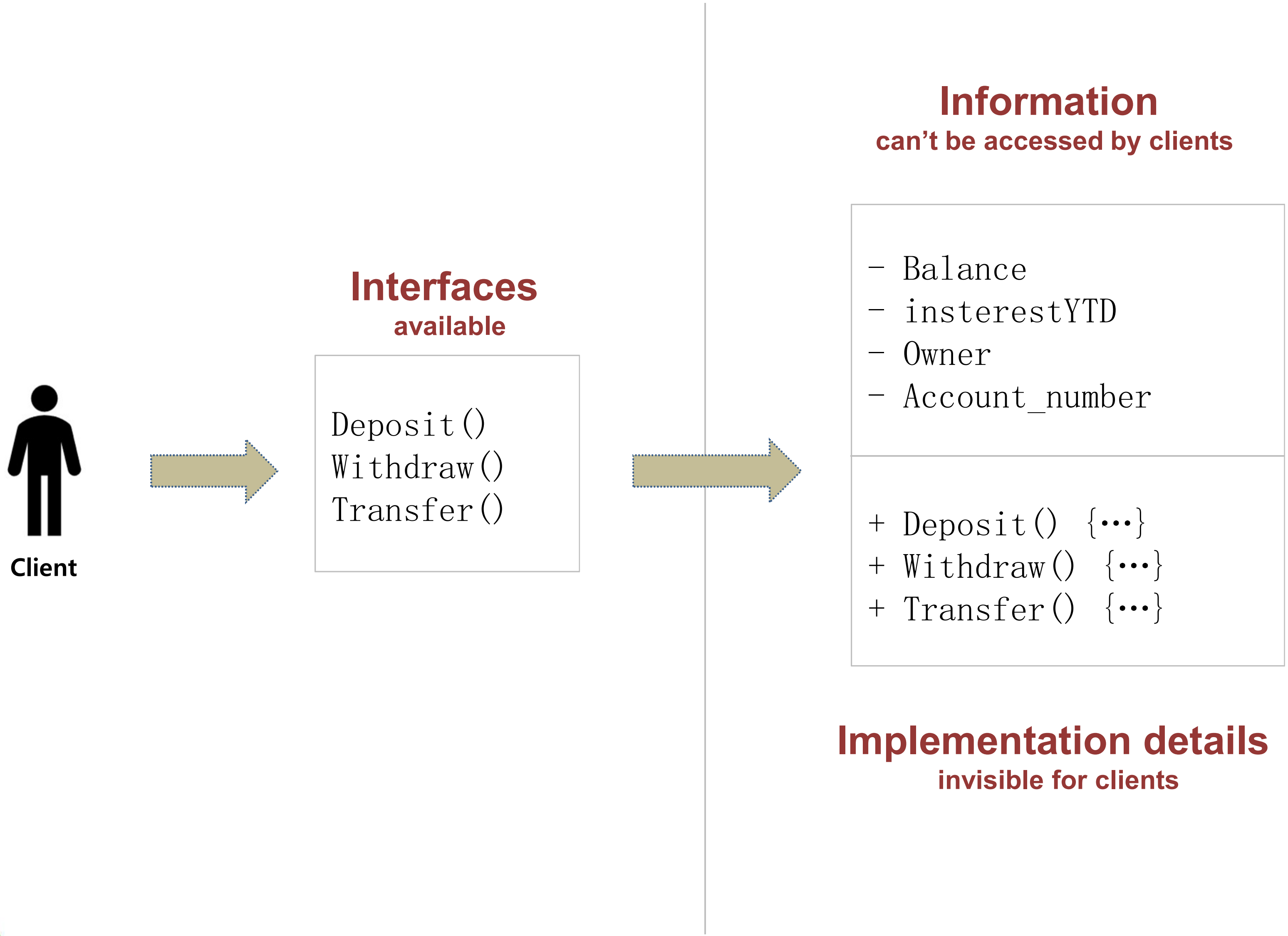
- Example:
  - When you drive a car, you don’t have know the details of how many cylinders the engine has or how the gasoline and air are mixed and ignited.
  - Instead you only have to know how to use the controls.

Example : Encapsulation

- Professor Yoo needs to be able to teach 4 classes in the next semester.



Encapsulation as Information Hiding



Client

Interfaces
available

```
Deposit()
Withdraw()
Transfer()
```

Information
can’t be accessed by clients

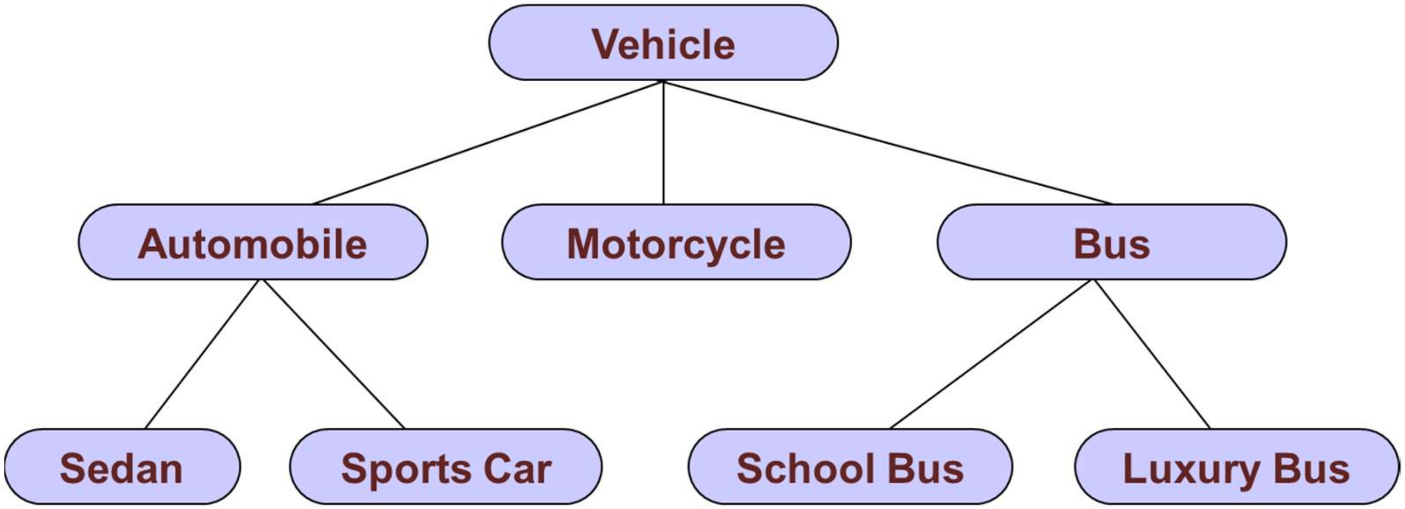
```
- Balance
- insterestYTD
- Owner
- Account_number
```

```
+ Deposit()  {…}
+ Withdraw()  {…}
+ Transfer()  {…}
```

Implementation details
invisible for clients

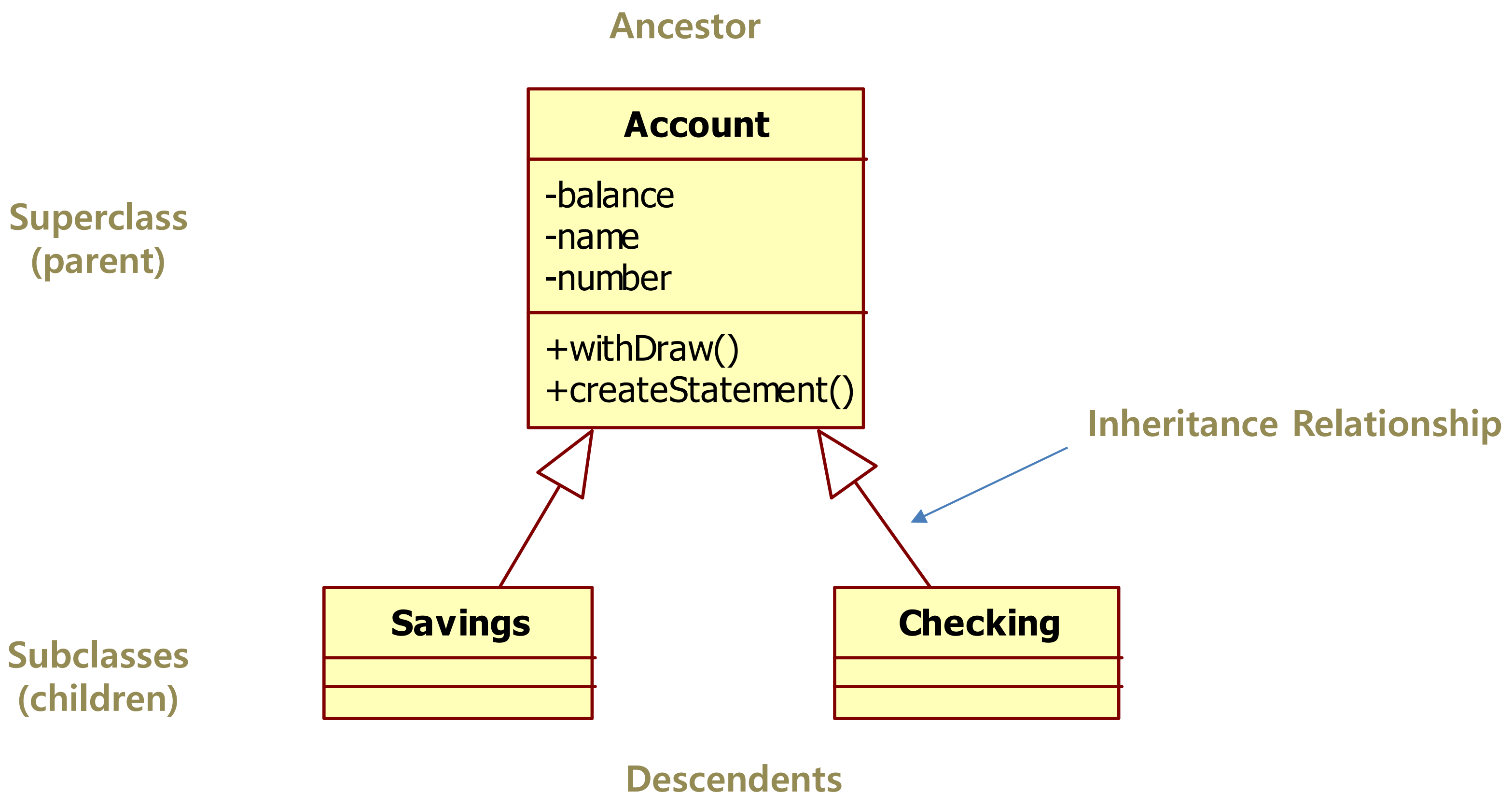
3. Inheritance

- **Inheritance** :
  - "is a kind of" , "is-a" relationship
  - A way of organizing classes
  - Classes with properties in common can be grouped so that their common properties are only defined once.



Example : Single Inheritance

- One class inherits from another.



Ancestor

Superclass
(parent)

Account

-balance
-name
-number

+withDraw()
+createStatement()

Inheritance Relationship

Subclasses
(children)

Savings

Checking

Descendents

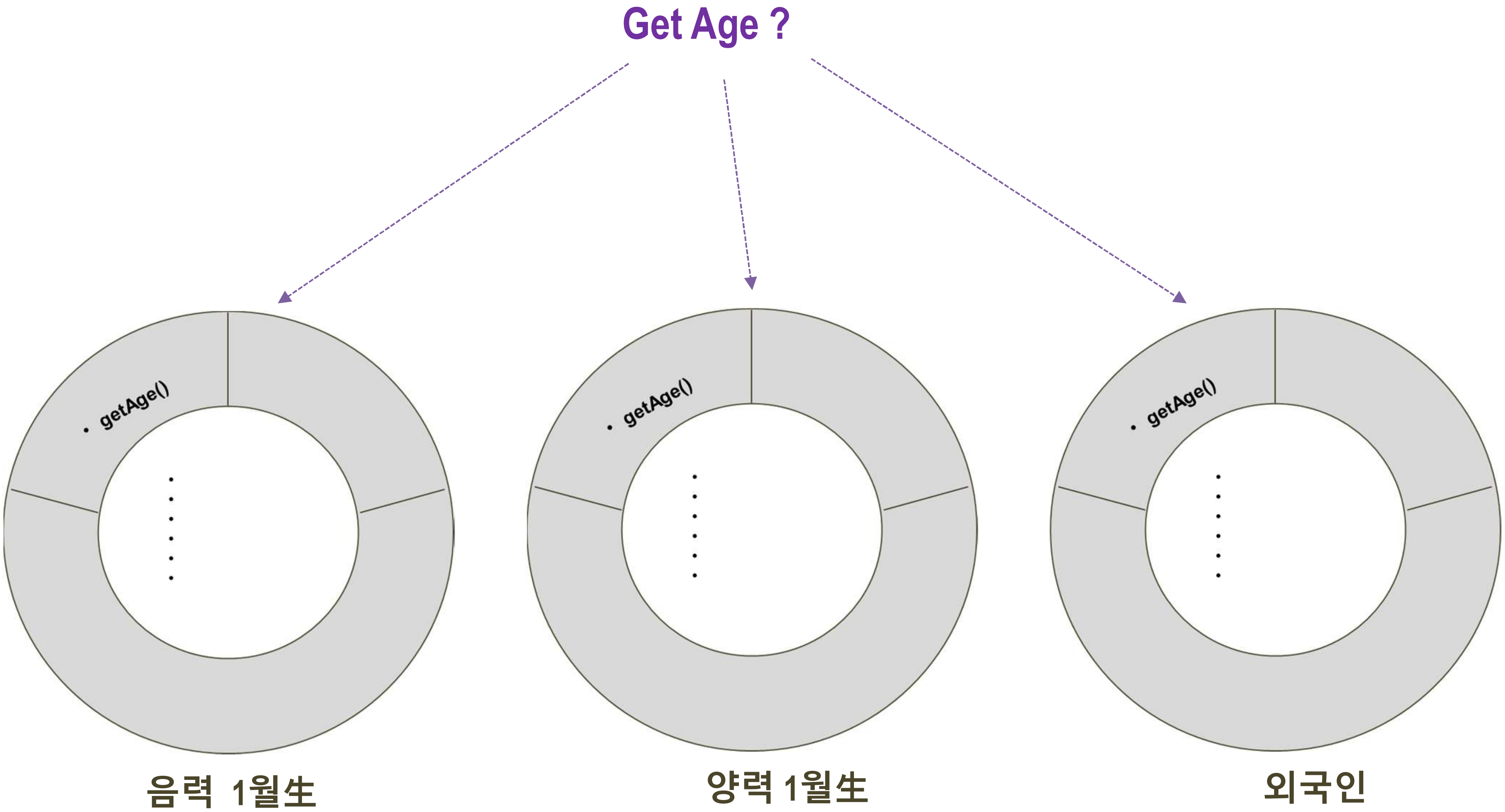
4. Polymorphism

- **Polymorphism** :
  - The ability to hide many different implementation behind a single interface.
  - The same word or phrase can mean different things in different contexts.

- Example:
  - In English, a bank can mean side of a river or a place to put money

- In Java,
  - Two or more classes could each have a method called `output`.
  - Each `output` method would do the right thing for the class that it was in.
    - One `output` might display a number, whereas a different one might display a name.

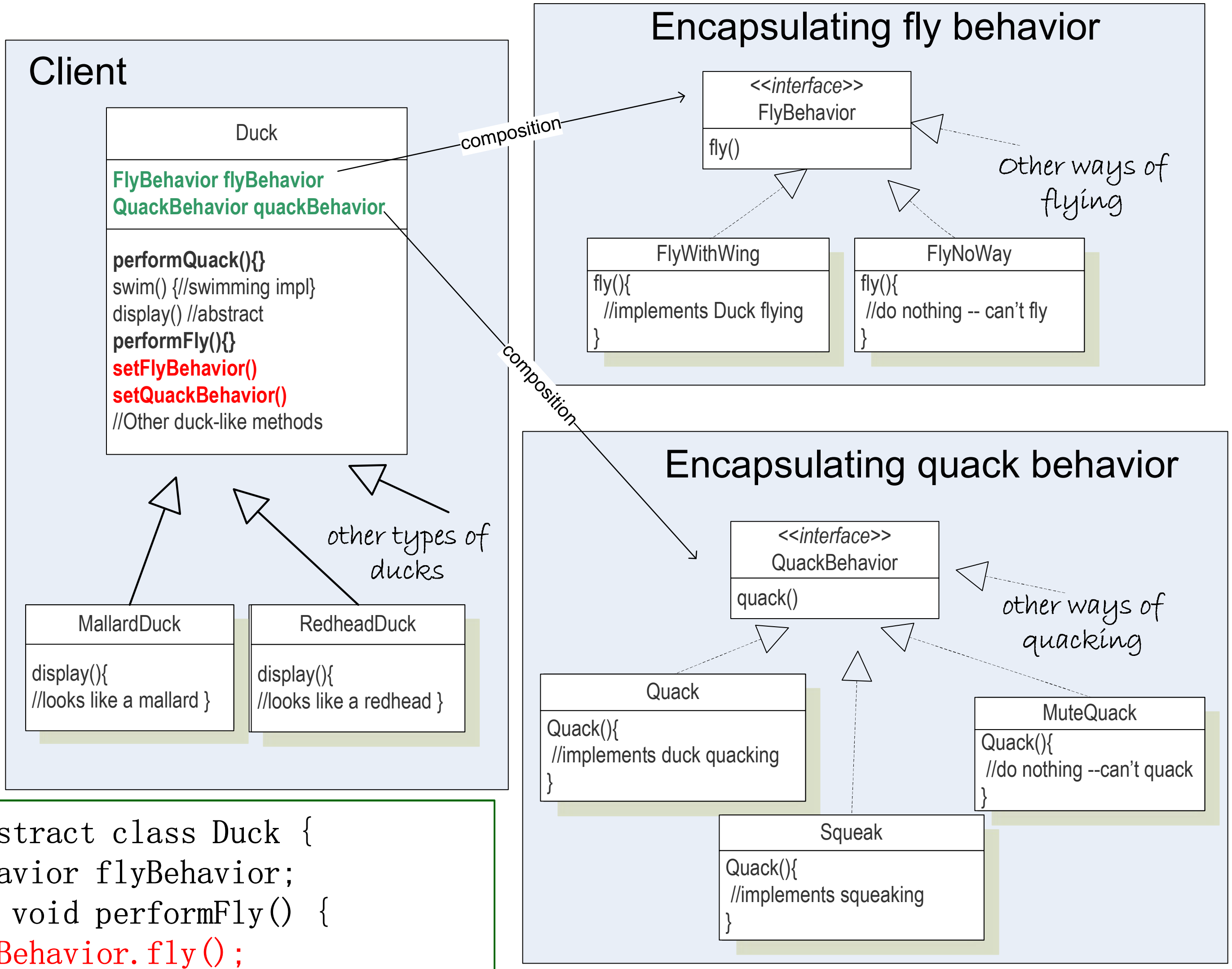
Example : Polymorphism



5. Composition

- **Object composition** :
  - "has_a" relationship between objects
  - Defined dynamically at runtime by acquiring references to other objects.
  - Does not break encapsulation, because objects are accessed solely through interfaces.
  - Any compatible object can be replaced with another at runtime.

Example : Composition



Client

Duck

FlyBehavior flyBehavior
QuackBehavior quackBehavior

performQuack(){}
swim() {//swimming impl}
display() //abstract
performFly(){}
setFlyBehavior()
setQuackBehavior()
//Other duck-like methods

other types of ducks

MallardDuck

display(){
//looks like a mallard }

RedheadDuck

display(){
//looks like a redhead }

composition

composition

Encapsulating fly behavior

<<interface>>
FlyBehavior

fly()

Other ways of flying

FlyWithWing

fly(){
 //implements Duck flying
}

FlyNoWay

fly(){
 //do nothing -- can't fly
}

Encapsulating quack behavior

<<interface>>
QuackBehavior

quack()

other ways of quacking

Quack

Quack(){
 //implements duck quacking
}

MuteQuack

Quack(){
 //do nothing --can't quack
}

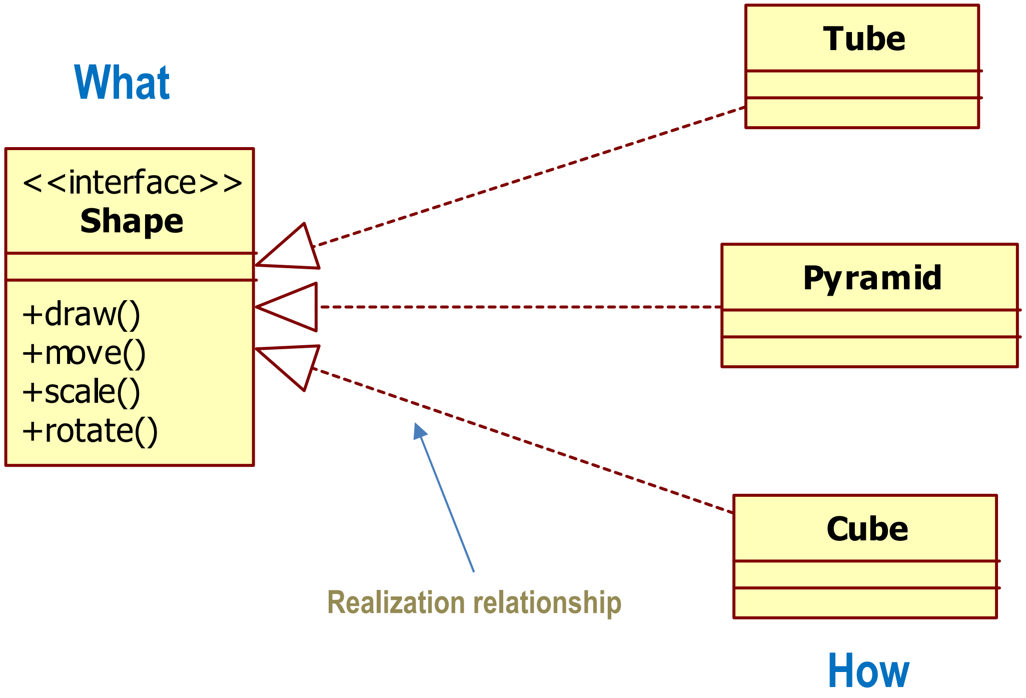
Squeak

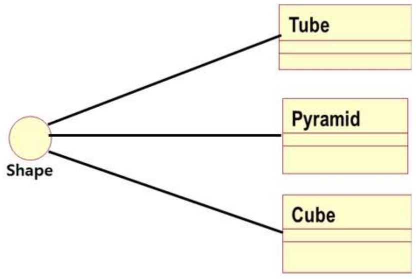
Quack(){
 //implements squeaking
}

```
public abstract class Duck {
   FlyBehavior flyBehavior;
   public void performFly() {
      flyBehavior.fly();
   }
}
```

6. Interface

- Interface
  - A collection of operations specifying a service of a class or component
  - Interfaces formalize polymorphism.
  - Interfaces support “plug-and-play” architectures.

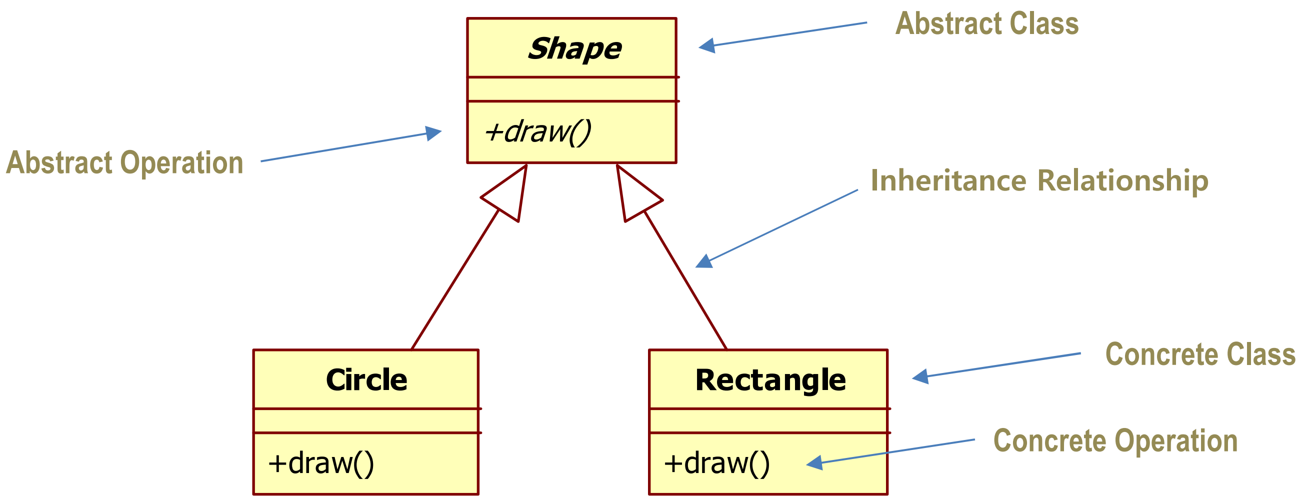




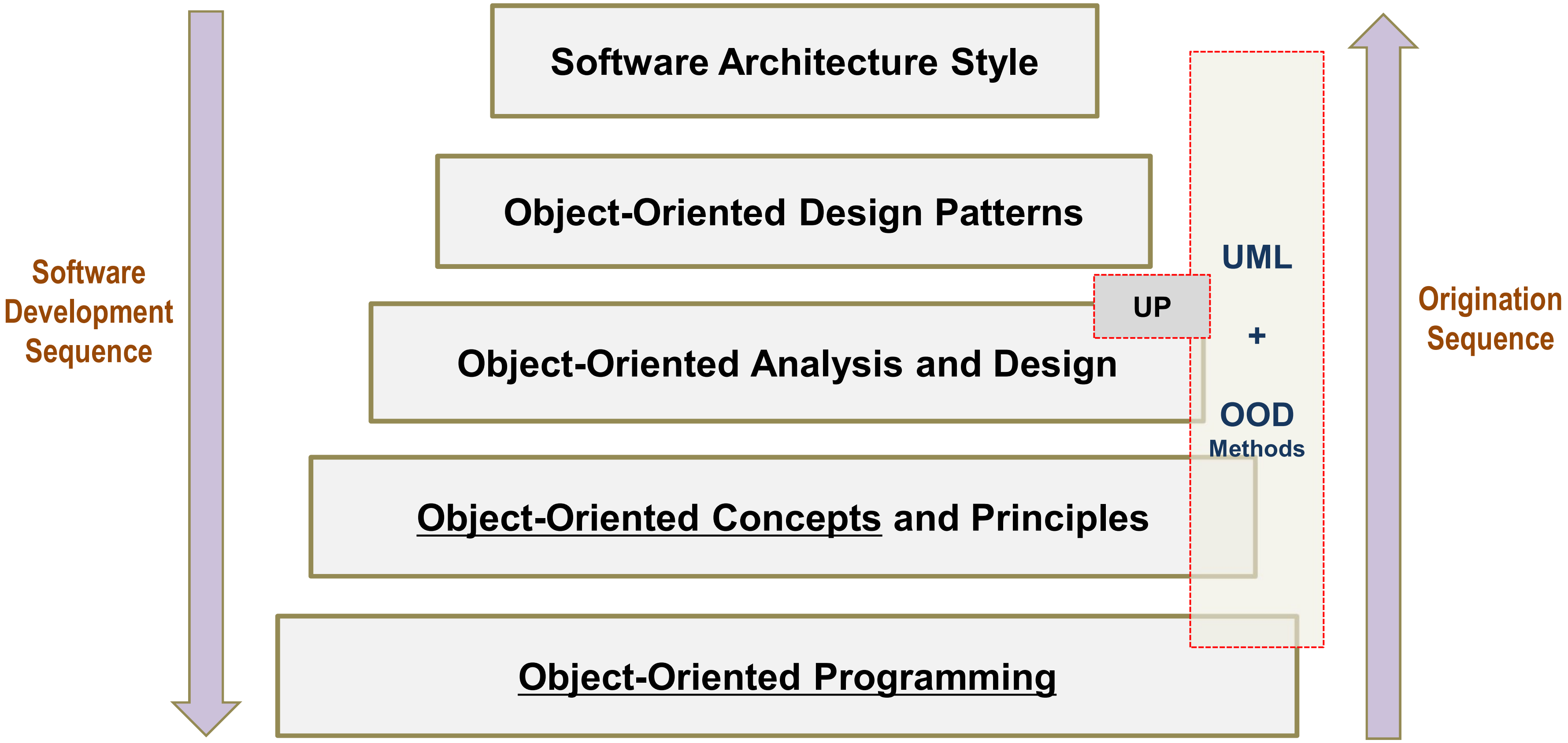
7. Abstract Class

- **Abstract class**
  - A class that may not has any direct instances.

- Abstract operation
  - An incomplete operation requiring a child to supply an implementation of the operation



An Overview of Object-Oriented Development



Software Development Sequence

Software Architecture Style

Object-Oriented Design Patterns

UP

Object-Oriented Analysis and Design

Object-Oriented Concepts and Principles

Object-Oriented Programming

UML

+

OOD Methods

Origination Sequence

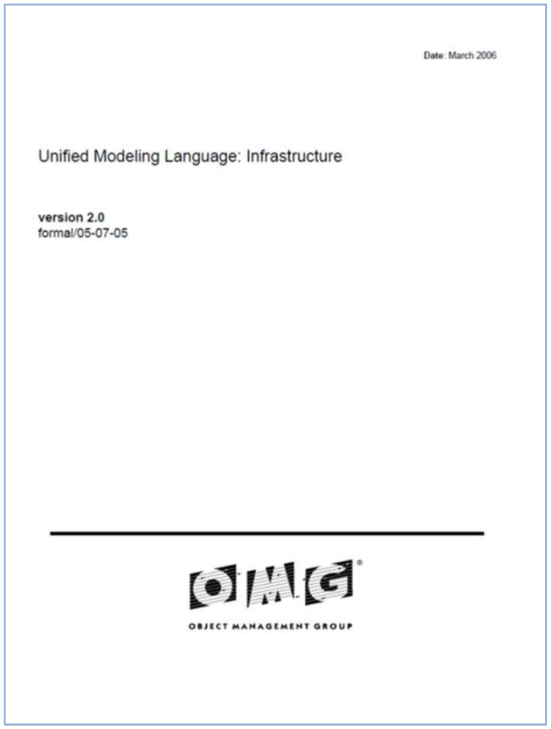
5 6

An Introduction to UML

UML

- **Unified Modeling Language** for
  - Visualizing , Specifying , Constructing and
  - Documenting the artifacts of software-intensive systems.
- Offer vocabulary and rules for **communication**
  - http://www.uml.org/
- Combine the best of the best from
  - Data Modeling (Entity Relationship Diagrams)
  - Business Modeling (work flow)
  - Object Modeling
  - Component Modeling (development and reuse - middleware, COTS)

de facto industry standard



Date: March 2006

Unified Modeling Language: Infrastructure

version 2.0
formal/05-07-05

OMG®
OBJECT MANAGEMENT GROUP

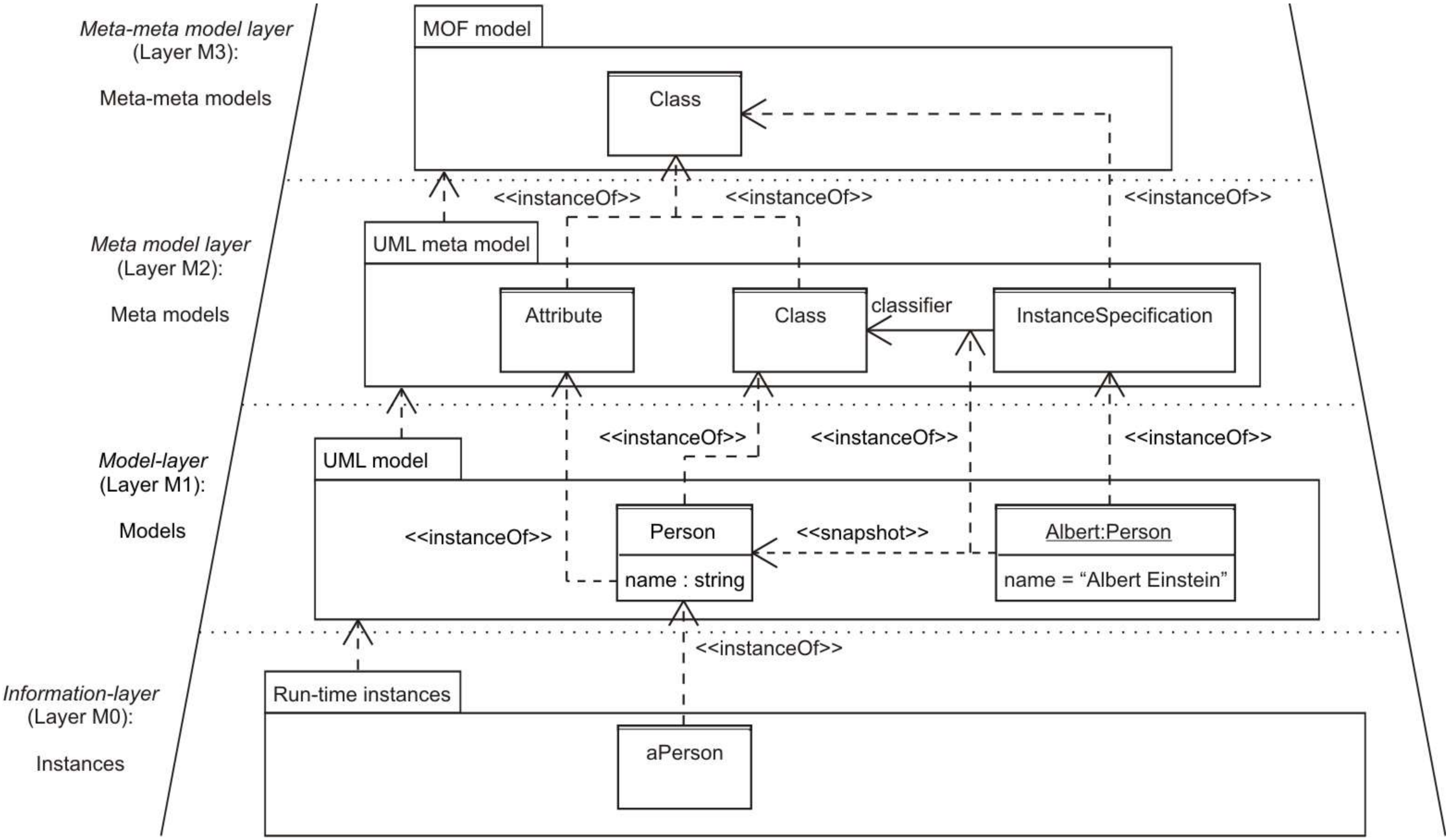
The UML Semantics

- **4-layer metamodel architecture**
  - **instance → model → meta model → meta-meta model**

- **MOF** **(Meta Object Facility)** defines a four-layer meta model hierarchy.
  - Layer M3: Meta-meta model layer **(The MOF model)**
  - Layer M2: Meta model layer **(The UML meta model)**
  - Layer M1: Model layer **(The UML model)**
  - Layer M0: Information layer **(the Application)**

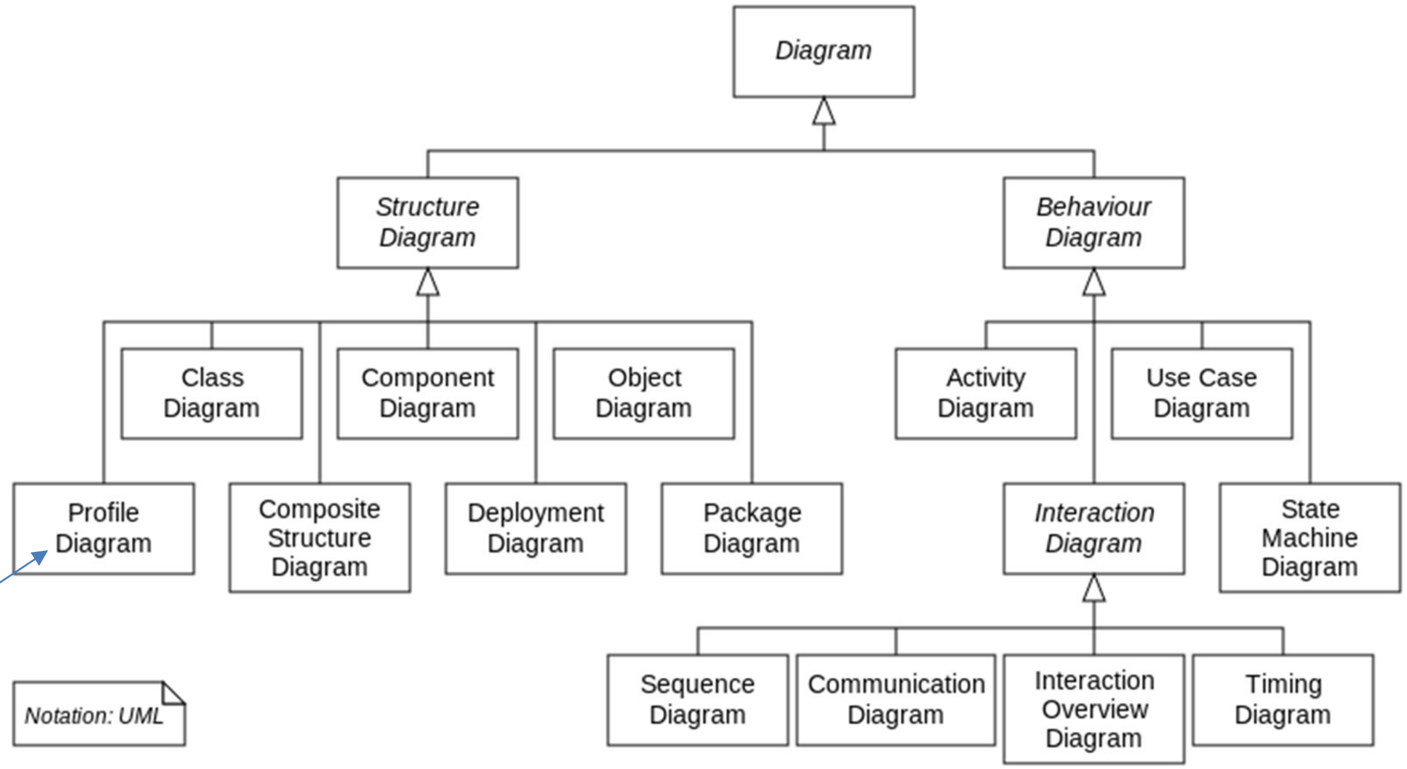
- MOF and UML are aligned.
  - The UML infrastructure contains all the concepts needed for the specification of UML and MOF.

The Meta Model Hierarchy of the MOF (for UML)



UML 2.0 Diagrams

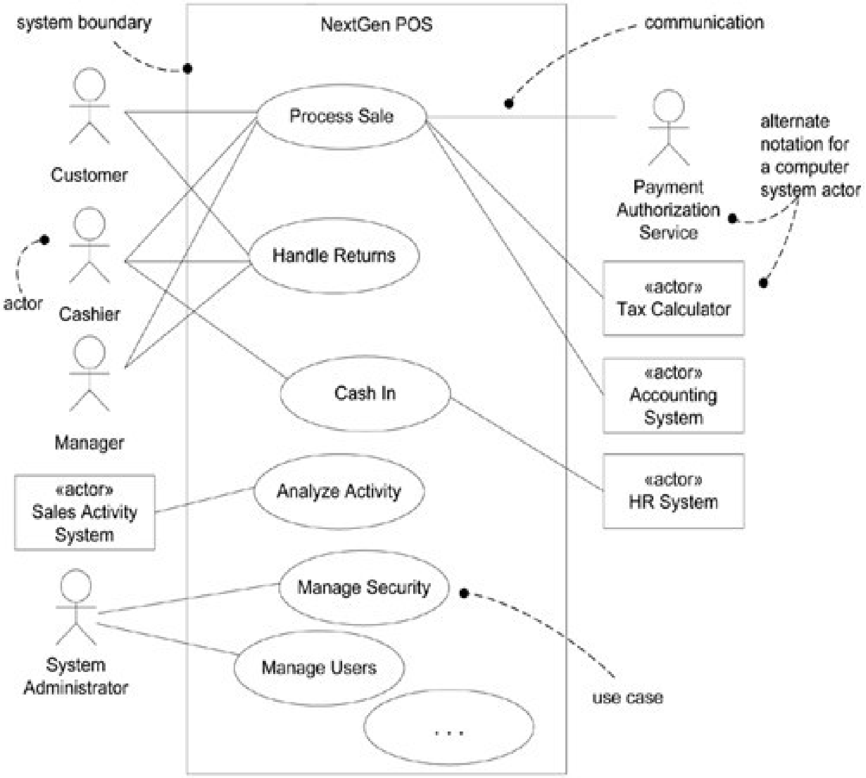
- **13 UML diagrams**



UML 2.2

1. Use Case Diagram

- **Use case diagram** illustrates the name of use cases and actors, and the relationships between them.
  - **Use case** : a collection of related success and failure scenarios, that describe how an actor uses the system to achieve a goal
  - **Actor** : something with behavior, such as a person, computer or organization



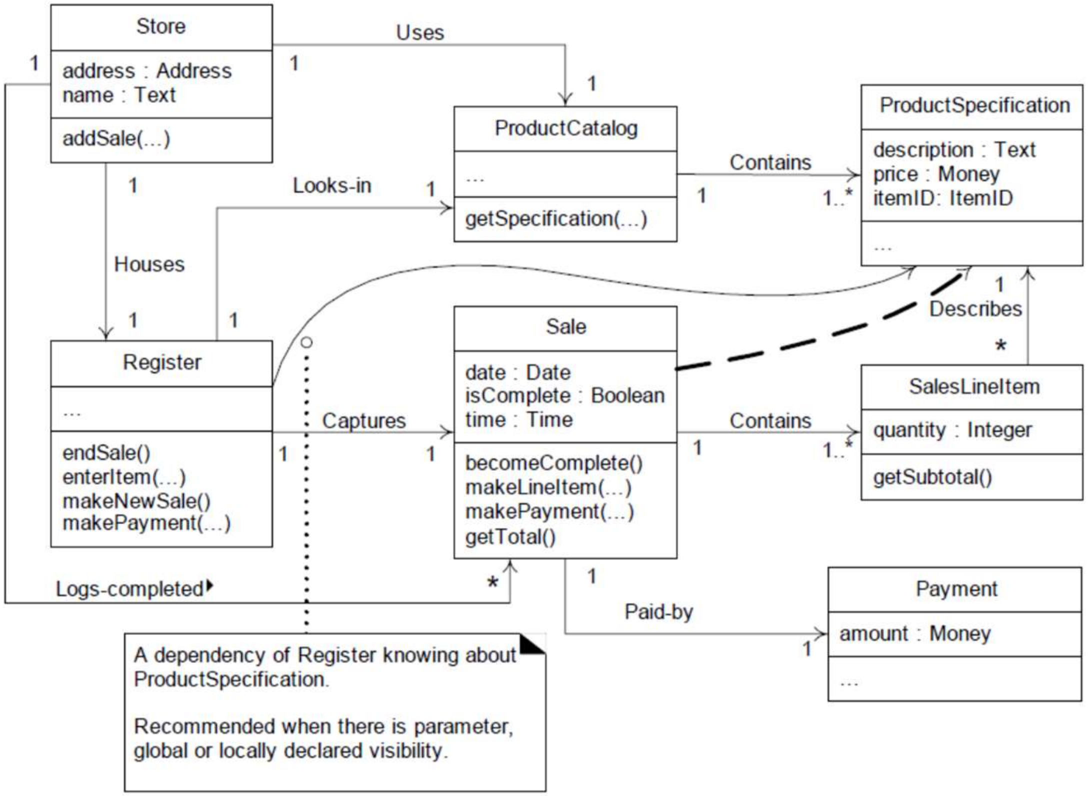
Use case: **Handle Returns**

Main Success Scenario:
- A customer arrives at a checkout with items to return.
- The cashier uses the POS system to record each returned item …

Alternate Scenarios:
- If the customer paid by credit, and the reimbursement transaction to their credit account is rejected, inform the customer and pay them with cash …

2. Class Diagram

- **Class diagrams** show the classes of the system, their inter-relationships, and the operations and attributes of the classes.
  - Domain model
  - Design class diagram (DCD)

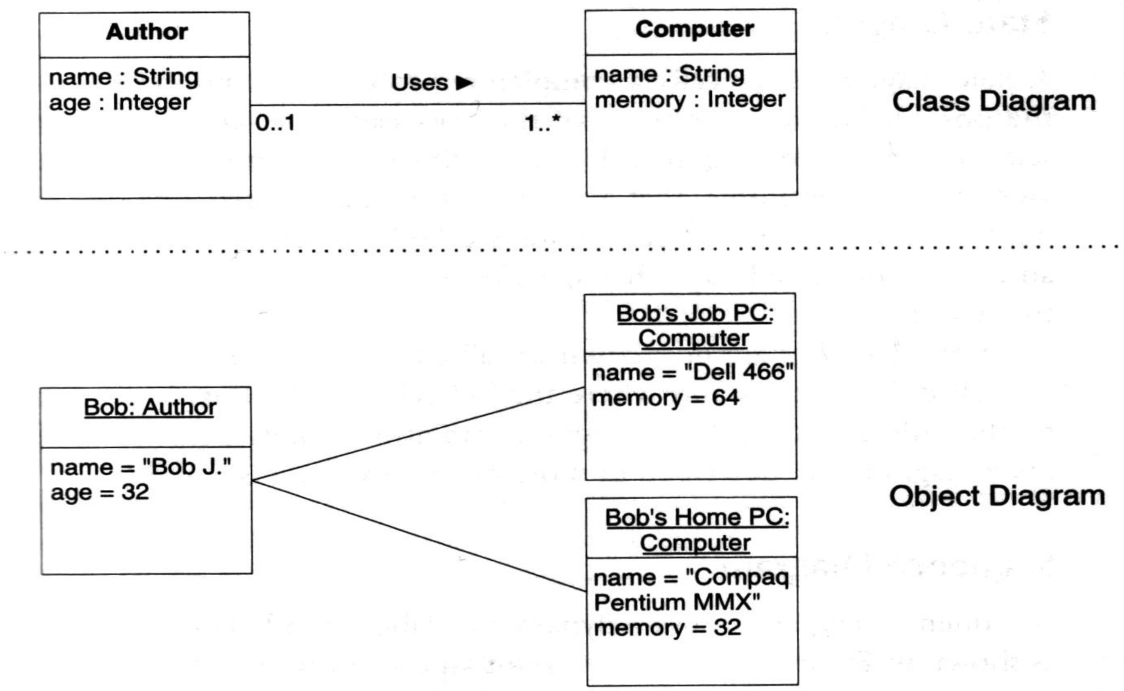


A dependency of Register knowing about ProductSpecification.

Recommended when there is parameter, global or locally declared visibility.

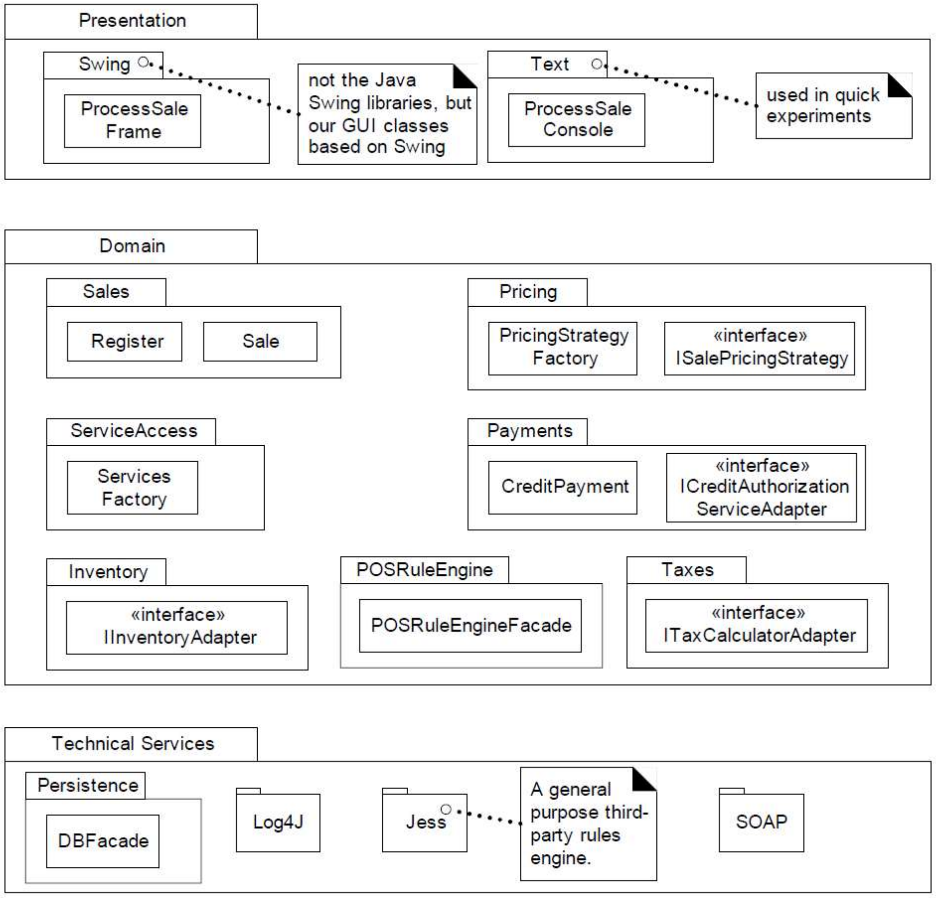
3. Object Diagram

- **Object diagrams** are useful for exploring real world examples of objects and the relationships between them.
  - Shows instances of classes at a specific point of time. (*i.e.,* snapshot)



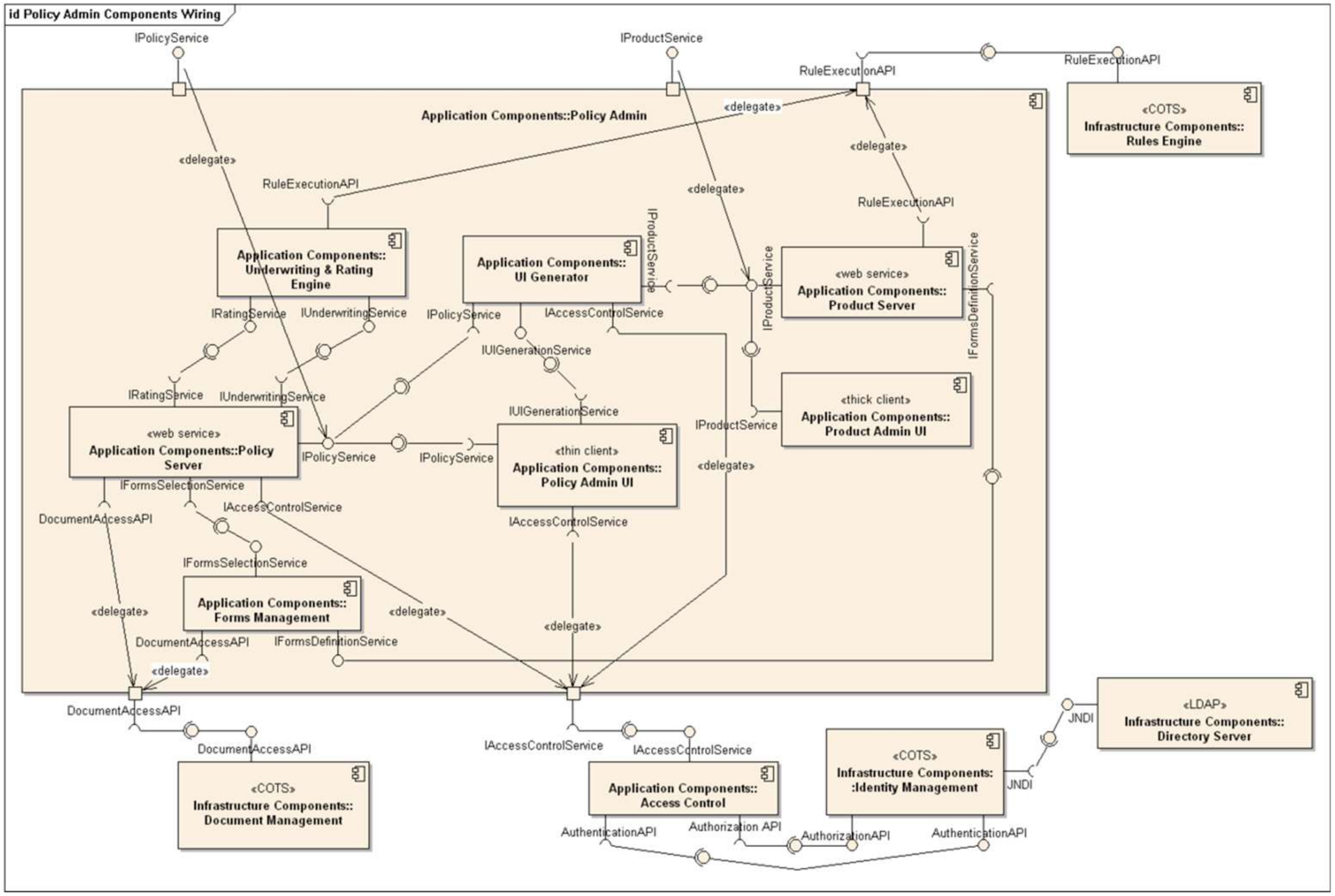
4. Package Diagram

- **Package diagrams** group classes into packages and simplify complex class diagrams.
  - A package is a collection of logically related UML elements.


Presentation
Swing
ProcessSale Frame
not the Java Swing libraries, but our GUI classes based on Swing
Text
ProcessSale Console
used in quick experiments
Domain
Sales
Register
Sale
Pricing
PricingStrategy Factory
«interface» ISalePricingStrategy
ServiceAccess
Services Factory
Payments
CreditPayment
«interface» ICreditAuthorization ServiceAdapter
Inventory
«interface» IInventoryAdapter
POSRuleEngine
POSRuleEngineFacade
Taxes
«interface» ITaxCalculatorAdapter
Technical Services
Persistence
DBFacade
Log4J
Jess
A general purpose third-party rules engine.
SOAP

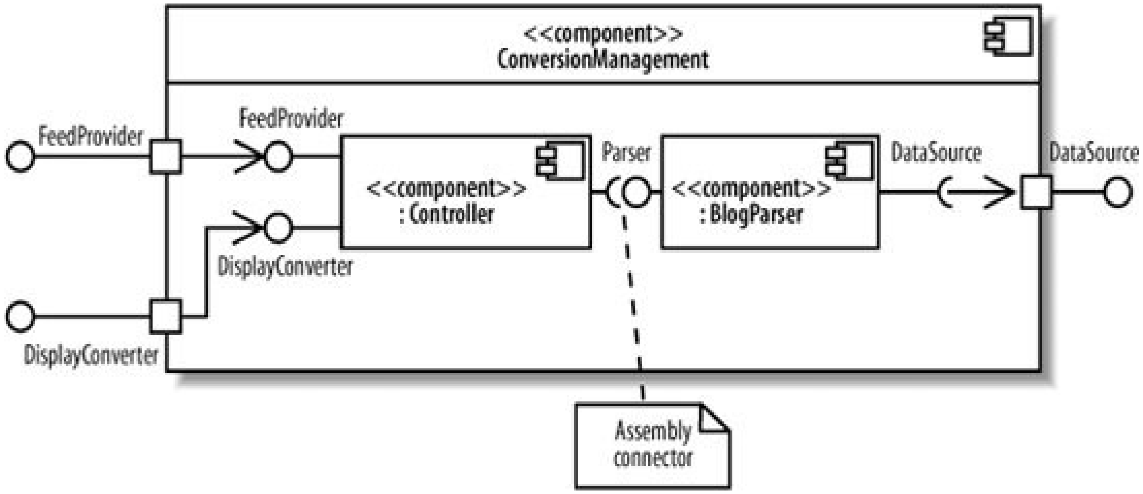
5. Component Diagram

- **Component diagrams** depicts how <u>components</u> are wired together to form larger components or software systems.
  - Illustrate the structure and inter-dependency of arbitrarily complex systems



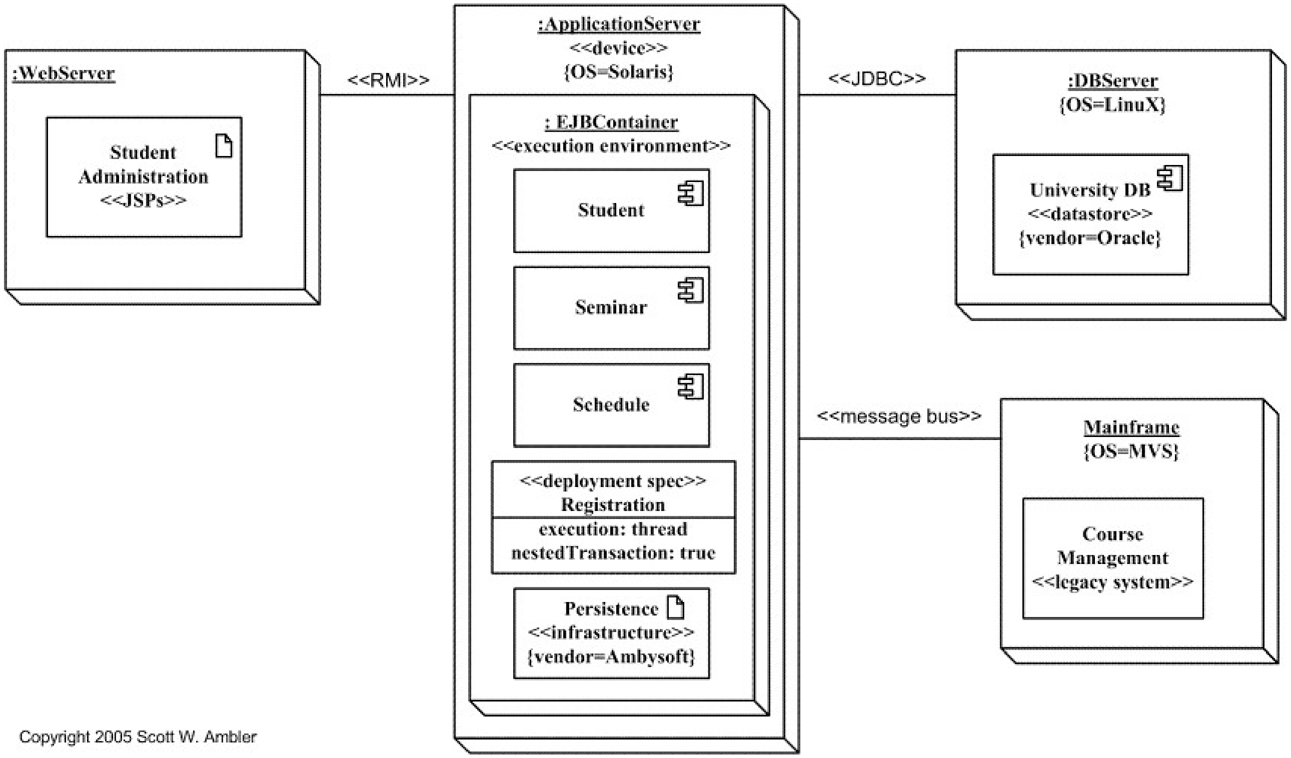
6. Composite Structure Diagram

- **Composite structure diagrams** are used to explore run-time instances of interconnected instances collaborating over communications links.
  - Show the internal structure (including parts and connectors) of components.



7. Deployment Diagram

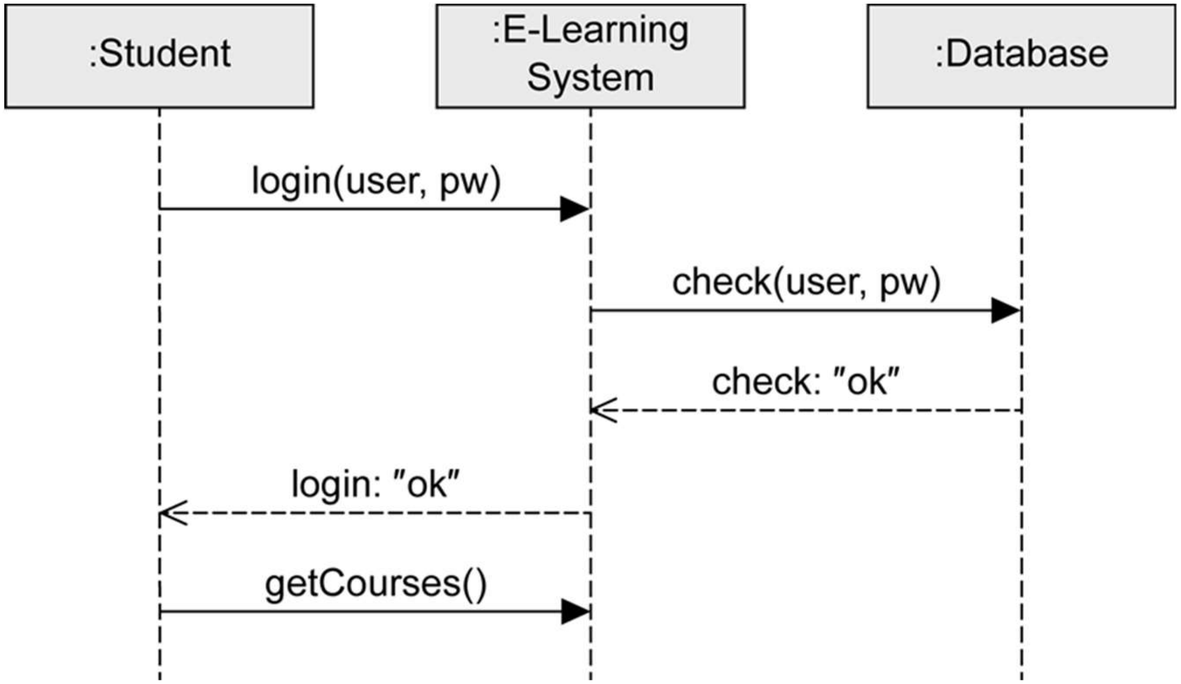
- **Deployment diagrams** depict a static view of the run-time configuration of hardware nodes and the software components running on those nodes.



Copyright 2005 Scott W. Ambler

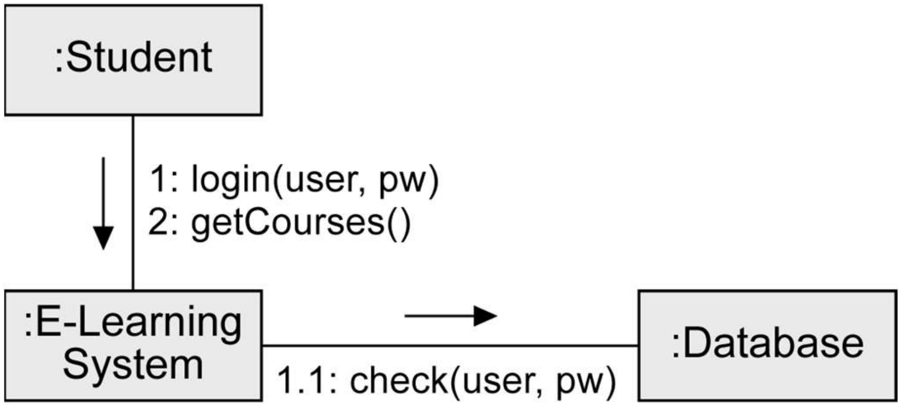
8. Sequence Diagram

- **Sequence diagrams** model the collaboration of objects based on a time sequence.
  - Show how the objects interact with others in a particular scenario of a use case.



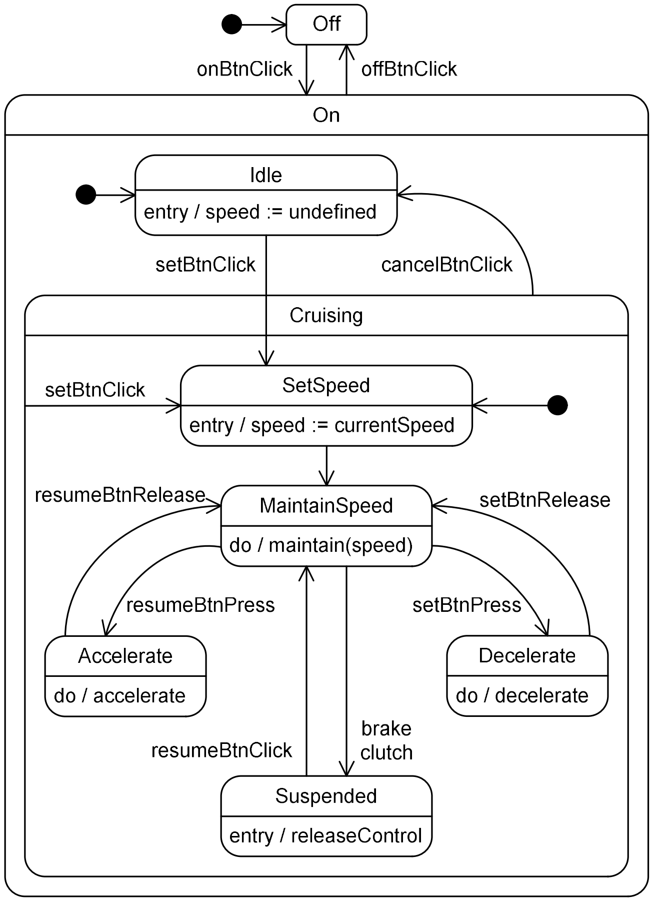
9. Communication Diagram

- **Communication diagrams** are used to model the dynamic behavior of the use case. (called collaboration diagram)
  - ≈ Sequence diagram
  - More focused on showing the collaboration of objects rather than the time sequence.



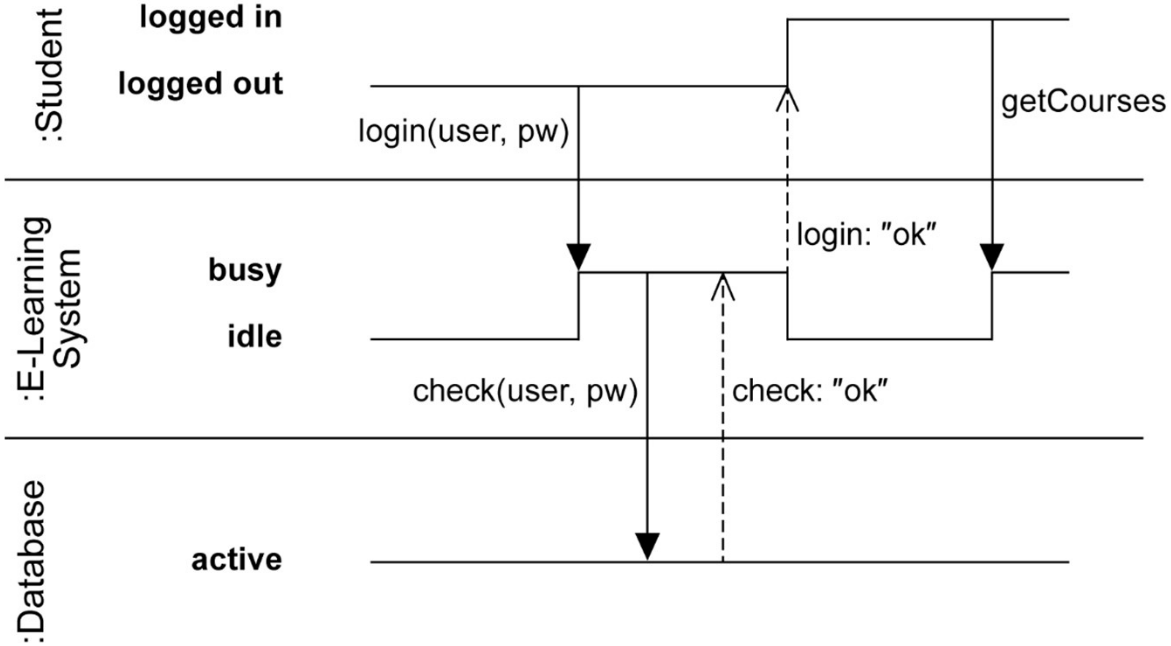
10. State (Statechart) Diagram

- **State diagrams** can show different states of an entity and how an entity responds to various events by changing from one state to another.
  - Originated from the Statechart formalism
  - The history of an entity is modeled by a finite state diagram.



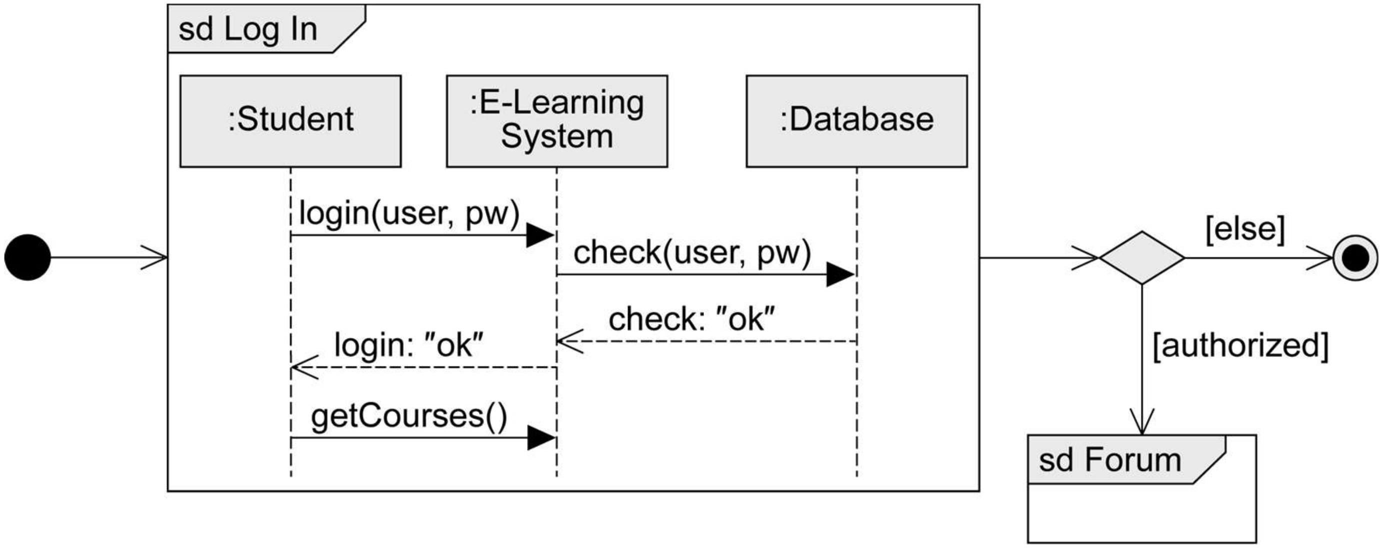
11. Timing Diagram

- **Timing diagrams** show the behavior of the objects in a given period of time.
  - A special form of a sequence diagram
  - The time increases from left to right and the lifelines are shown in separate compartments arranged vertically.



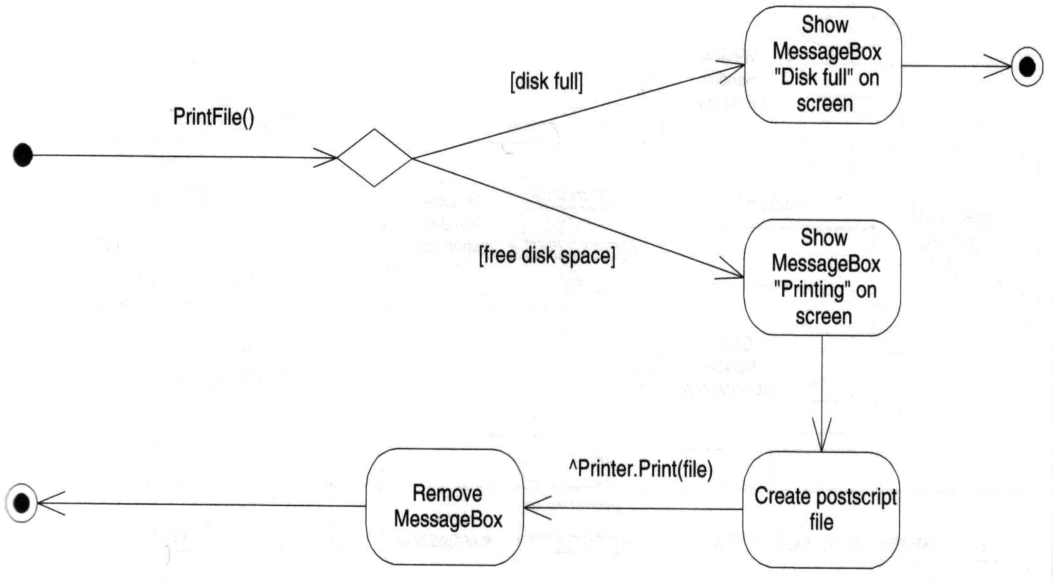
12. Interaction Overview Diagram

- **Interaction overview diagrams** focus on the overview of the flow of control of the interactions.
  - A variant of the Activity Diagram, where the nodes are the interactions or interaction occurrences.

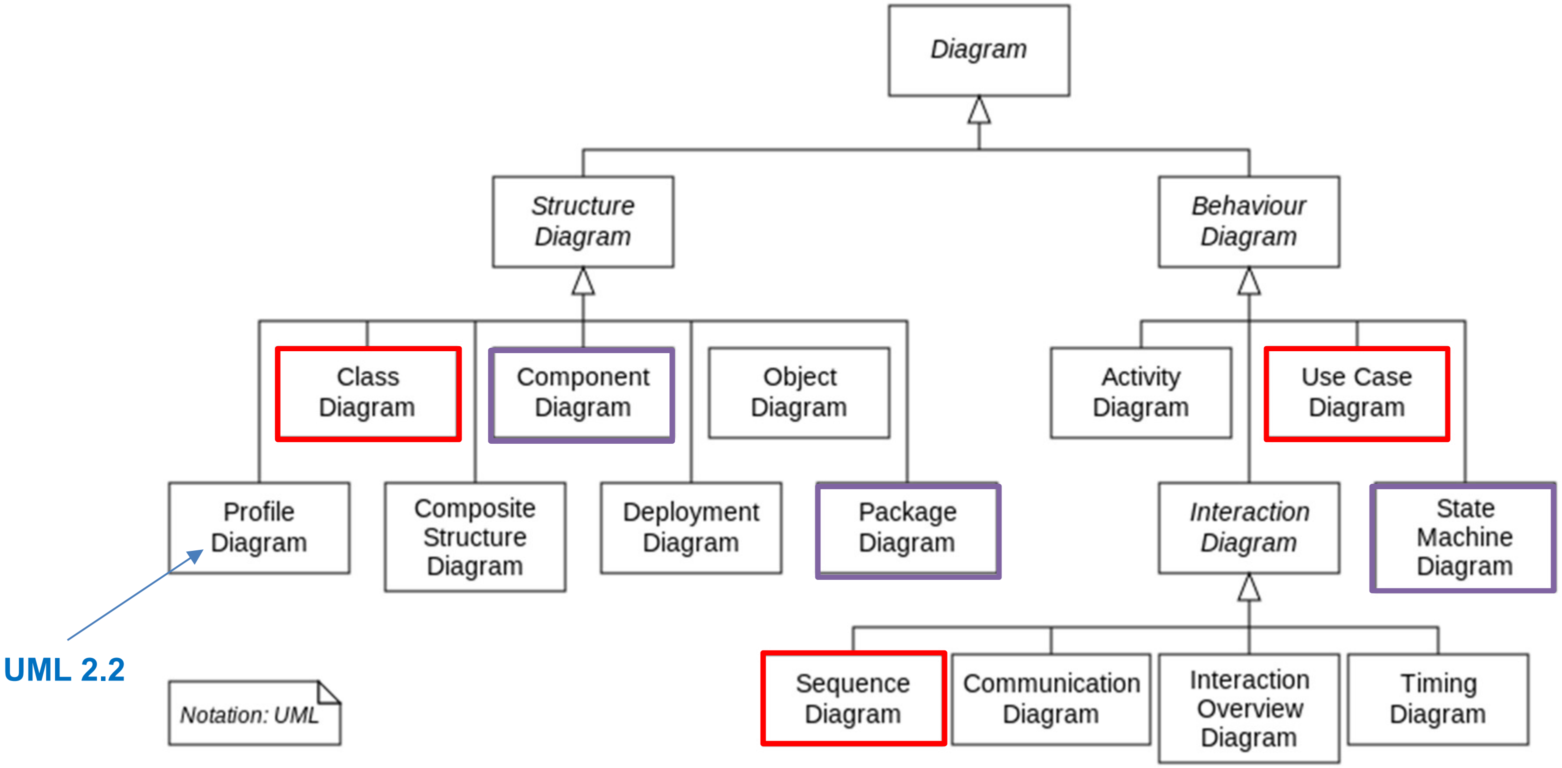


13. Activity Diagram

- **Activity diagrams** help to describe the flow of control of the target system.
  - Exploring complex business rules and operations, describing the use case and the business process.
  - It is an object-oriented equivalent of flow-charts and DFDs (data flow diagrams).



13 UML Diagrams



Dependable Software Laboratory