

# COMP6115

## Object Oriented Analysis and Design

### Session #8

The background is a solid blue color. On the left side, there are two overlapping circles of a lighter blue shade. One circle is positioned higher and further to the left, while the other is lower and further to the right, creating a lens-like intersection in the center-left area.

# **Class and Method Design**

# Learning Outcomes

LO1: Identify the basic concept of advance topic in Object Oriented Analysis and Design

LO2 : Use the knowledge to develop documentation for object oriented software analysis and design using Unified Modelling Language

LO3 : Analyze any problem in any software application and find out the alternative solutions using object oriented analysis and design approach

## Chapter 8:

# Class and Method Design

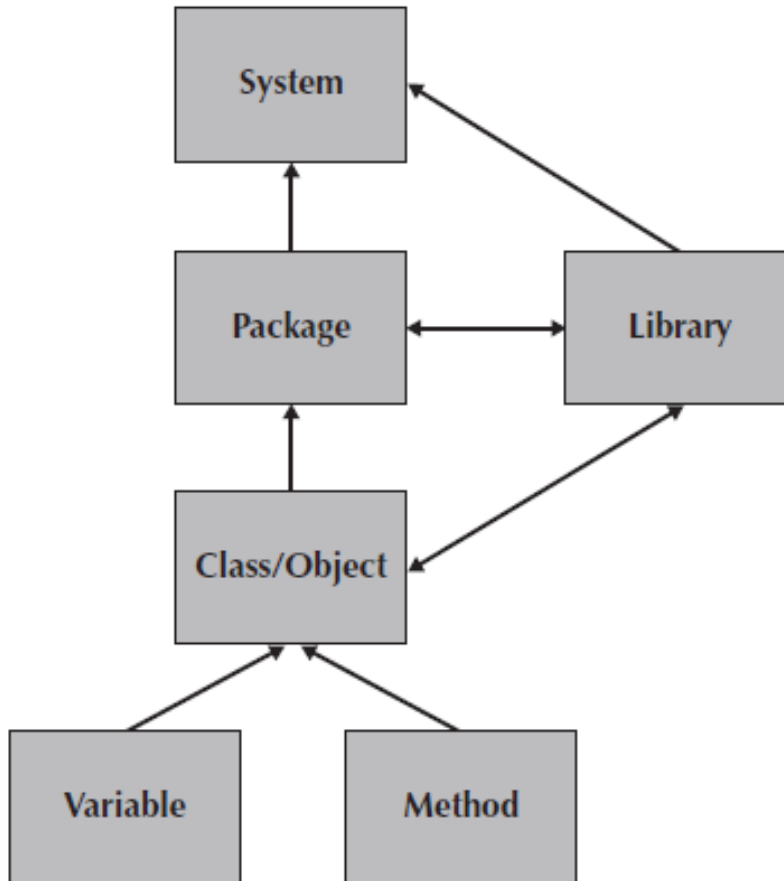
# Objectives

- Become familiar with **coupling**, **cohesion**, and connascence.
- Be able to specify, restructure, and optimize object designs.
- Be able to identify the reuse of predefined classes, libraries, frameworks, and components.
- Be able to specify constraints and contracts.
- Be able to create a method specification.

# Introduction

- Review the characteristics of object orientation
- Present useful criteria for evaluating a design
- Present design activities for classes and methods
- Present the concept of constraints & contracts to define object collaboration
- Discuss how to specify methods to augment method design
- Caution:
  - Class & method design must precede coding
  - While classes are specified in some detail, jumping into coding without first designing them may be disastrous

# Levels of Abstraction in Object-Oriented Systems



Source: Based on material from David P. Tegarden, Steven D. Sheetz, and David E. Monarchi, "A Software Complexity Model of Object-Oriented Systems," *Decision Support Systems* 13 (March 1995): 241–262.

# Core packages in Java SE 8

- `java.lang` — basic language functionality and fundamental types
- `java.util` — collection data structure classes
- `java.io` — file operations
- `java.math` — multiprecision arithmetics
- `java.nio` — the Non-blocking I/O framework for Java
- `java.net` — networking operations, sockets, DNS lookups, ...
- `java.security` — key generation, encryption and decryption
- `java.sql` — Java Database Connectivity (JDBC) to access databases
- `java.awt` — basic hierarchy of packages for native GUI components
- `java.text` — Provides classes and interfaces for handling text, dates, numbers, and messages in a manner independent of natural languages.
- `java.rmi` — Provides the RMI package.
- `java.time` — The main API for dates, times, instants, and durations.
- `java.beans` — The `java.beans` package contains classes and interfaces related to JavaBeans components.
- `java.applet` — This package provides classes and methods to create and communicate with the applets.



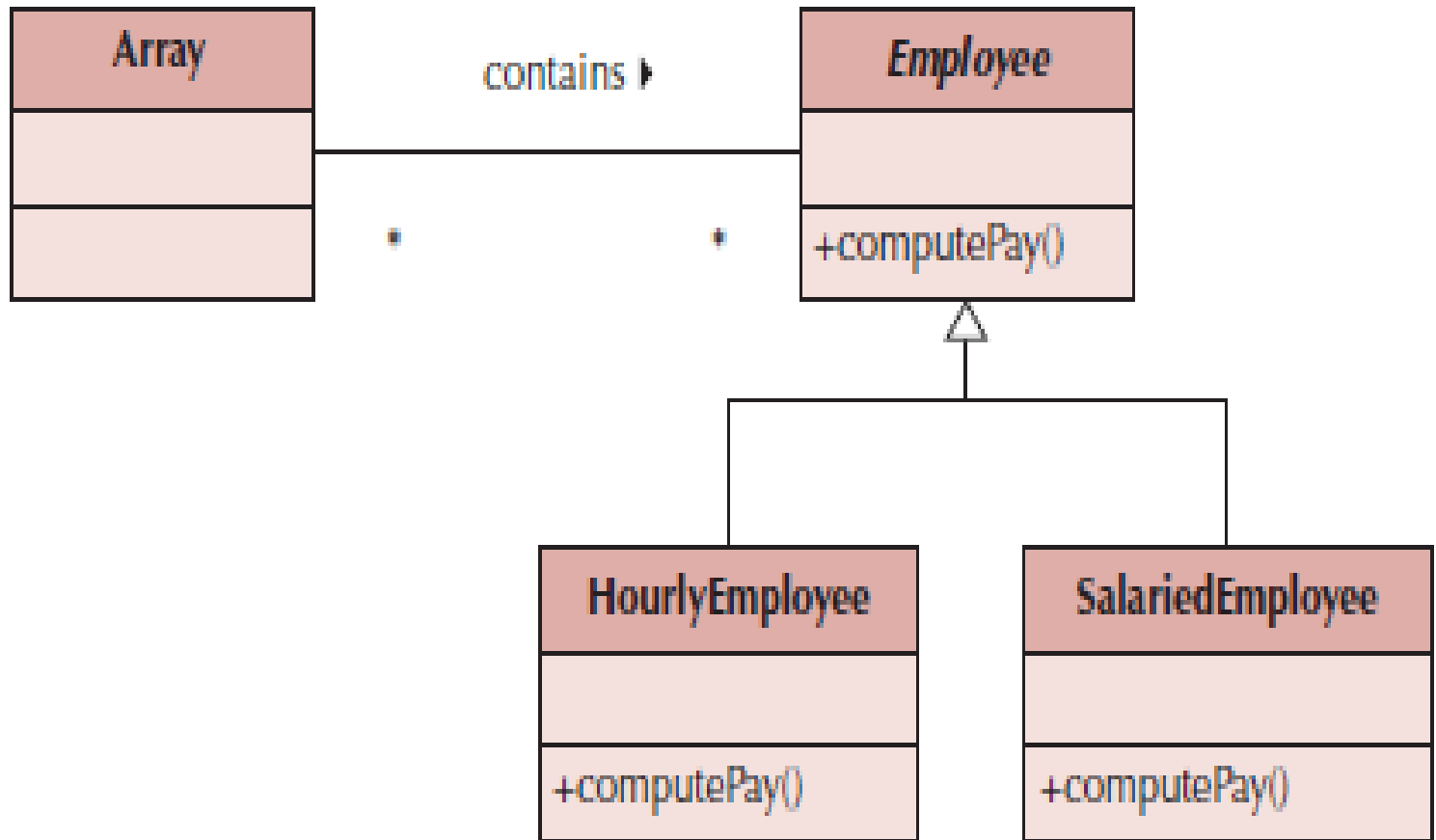
# Characteristics of OOSAD

- **Classes**
  - Instantiated classes are **objects**
  - Classes are defined with **attributes, states & methods**
  - Classes communicate through **messages**
- **Encapsulation & information hiding**
  - Combine data and operations into a single object
  - Reveal only how to make use of an object to other objects
  - Key to **reusability**
- **Polymorphism & dynamic binding**
- **Inheritance**

# Polymorphism & Dynamic Binding

- Polymorphism
  - The ability to take on several different forms
    - **Over-riding (inheritance): Implement all methods in an interface**
    - **Function over-loading: same name, different function content**
      - `plotGraph(x,y)`
      - `plotGraph(x,y,z)`
      - >> diff # of parameter, diff type of parameter, diff order or parameter
    - **Templates? LinkedList in Java LL list<E>**
      - **Swap function -> define dynamic data type in the run!**
    - Same message triggers different methods in different objects
  - **Dynamic binding vs Static Binding**
    - Methods—the specific method used is selected at run time
    - Attributes—data type is chosen at run time
    - Implementation of dynamic binding is language specific
  - Decisions made at run time may induce run-time errors
  - Need to ensure semantic consistency

# Polymorphism Example

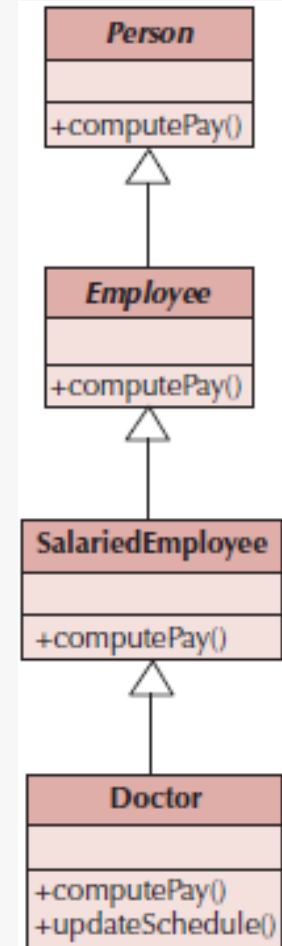


# Inheritance

- Permits reuse of existing classes with extensions for new attributes or operations
- Types
  - Single inheritance -- one parent class
  - Multiple inheritance -- multiple parent classes (not supported by all programming languages)
  - Redefinition of methods and/or attributes
    - Not supported by all programming languages
    - May cause inheritance conflict
- Designers must know what the chosen programming language supports

# Inheritance Conflicts

- An attribute or method in a sub-class with the same name as an attribute or method in the super class
- Cause is poor classification of sub-classes:
  - Generalization semantics are violated, or
  - Encapsulation and information hiding principle is violated
- May also occur in cases of multiple inheritance



# Design Criteria

- **A set of metrics to evaluate the design**
- **Coupling**—refers to the degree of the closeness of the relationship between classes
- **Cohesion**—refers to the degree to which attributes and methods of a class support a single object
  - Class Cohesion
    - Class Student: NIM, Name, GPA 👍
    - Class Student: NIM, Name, GPA, **Lecturer Name** 👎
- **Connascence**—refers to the degree of interdependency between objects

# Coupling

- **Close coupling means that changes in one part of the design may require changes in another part**
- **Types**
  - **Interaction coupling measured through message passing**
  - **Inheritance coupling deals with the inheritance hierarchy of classes**
- **Minimize interaction coupling by restricting messages (Law of Demeter)**
- **Minimize inheritance coupling by using inheritance to support only generalization/specialization and the principle of substitutability**

# Law of Demeter

Messages should be sent only by an object:

to itself

to objects contained in attributes of itself or a superclass

to an object that is passed as a parameter to the method

to an object that is created by the method

to an object that is stored in a global variable



# Types of Interaction Coupling

Level	Type	Description
Good	No Direct Coupling	The methods do not relate to one another; that is, they do not call one another.
↓	Data	The calling method passes a variable to the called method. If the variable is composite (i.e., an object), the entire object is used by the called method to perform its function.
	Stamp	The calling method passes a composite variable (i.e., an object) to the called method, but the called method only uses a portion of the object to perform its function.
	Control	The calling method passes a control variable whose value will control the execution of the called method.
	Common or Global	The methods refer to a "global data area" that is outside the individual objects.
Bad	Content or Pathological	A method of one object refers to the inside (hidden parts) of another object. This violates the principles of encapsulation and information hiding. However, C++ allows this to take place through the use of "friends."

Source: These types were adapted from Meilir Page-Jones, *The Practical Guide to Structured Systems Design*, 2nd ed. (Englewood Cliffs, NJ: Yardon Press, 1988); and Glenford Myers, *Composite/Structured Design* (New York: Van Nostrand Reinhold, 1978).

# Cohesion

- A cohesive class, object or method refers to a single thing
- Types
  - Method cohesion
    - Does a method perform more than one operation?
    - Performing more than one operation is more difficult to understand and implement
  - Class cohesion
    - Do the attributes and methods represent a single object?
    - Classes should not mix class roles, domains or objects
  - Generalization/specialization cohesion
    - Classes in a hierarchy should show “a-kind-of” relationship, not associations or aggregations

# Types of Method Cohesion

Level	Type	Description
Good	Functional	A method performs a single problem-related task (e.g., calculate current GPA).
	Sequential	The method combines two functions in which the output from the first one is used as the input to the second one (e.g., format and validate current GPA).
	Communicational	The method combines two functions that use the same attributes to execute (e.g., calculate current and cumulative GPA).
	Procedural	The method supports multiple weakly related functions. For example, the method could calculate student GPA, print student record, calculate cumulative GPA, and print cumulative GPA.
	Temporal or Classical	The method supports multiple related functions in time (e.g., initialize all attributes).
	Logical	The method supports multiple related functions, but the choice of the specific function is chosen based on a control variable that is passed into the method. For example, the called method could open a checking account, open a savings account, or calculate a loan, depending on the message that is send by its calling method.
Bad	Coincidental	The purpose of the method cannot be defined or it performs multiple functions that are unrelated to one another. For example, the method could update customer records, calculate loan payments, print exception reports, and analyze competitor pricing structure.

Source: These types were adapted from Page-Jones, *The Practical Guide to Structured Systems*, and Myers, *Composite/Structured Design*.

# Types of Class Cohesion

Level	Type	Description
Good	Ideal	The class has none of the mixed cohesions.
↓	Mixed-Role	The class has one or more attributes that relate objects of the class to other objects on the same layer (e.g., the problem domain layer), but the attribute(s) have nothing to do with the underlying semantics of the class.
	Mixed-Domain	The class has one or more attributes that relate objects of the class to other objects on a different layer. As such, they have nothing to do with the underlying semantics of the thing that the class represents. In these cases, the offending attribute(s) belongs in another class located on one of the other layers. For example, a port attribute located in a problem domain class should be in a system architecture class that is related to the problem domain class.
Worse	Mixed-Instance	The class represents two different types of objects. The class should be decomposed into two separate classes. Typically, different instances only use a portion of the full definition of the class.

*Source: Page-Jones, Fundamentals of Object-Oriented Design in UML.*

# Cohesion & Coupling by PDM-LA02



<https://flipgrid.com/+85y3vzfe>

Mixtapes > Cohesion & Coupling by PDM-LA02

May 5, 2021

## Cohesion & Coupling by PDM-LA02

Active ▾

Share

Actions ▾





36 videos • 978 views

In these videos, our class LA02 - Program Design Methods explains the definition and examples of Cohesion & Coupling. We hope to increase the learning experience on this subject and study further by sharing our knowledge. Thank you! 😊🙏

#socialLearning #empowerEveryVoice #binusUniversity

Join Code: +85y3vzfe

### 36 Videos

Name	Date	Link			
 <b>JEREMIAH JASON</b>	Oct 9, 2020	<a href="#">+85y3vzfe/4ae1f9e9</a>	Share	Actions ▾	Remove
 <b>DIVEN C</b>	Oct 9, 2020	<a href="#">+85y3vzfe/8f851d02</a>	Share	Actions ▾	Remove

# Connascence

- Classes are so interdependent that a change in one necessitates a change in the other
- Good programming practice should:
  - Minimize overall connascence; however, when combined with encapsulation boundaries, you should:
    - Minimize across encapsulation boundaries (less interdependence between or among classes)
    - Maximize within encapsulation boundary (greater interdependence within a class)
  - A sub-class should never directly access any hidden attribute or method of a super class

# Types of Connascence

Type	Description
Name	If a method refers to an attribute, it is tied to the name of the attribute. If the attribute's name changes, the content of the method will have to change.
Type or Class	If a class has an attribute of type A, it is tied to the type of the attribute. If the type of the attribute changes, the attribute declaration will have to change.
Convention	A class has an attribute in which a range of values has a semantic meaning (e.g., account numbers whose values range from 1000 to 1999 are assets). If the range would change, then every method that used the attribute would have to be modified.
Algorithm	Two different methods of a class are dependent on the same algorithm to execute correctly (e.g., insert an element into an array and find an element in the same array). If the underlying algorithm would change, then the insert and find methods would also have to change.
Position	The order of the code in a method or the order of the arguments to a method is critical for the method to execute correctly. If either is wrong, then the method will, at least, not function correctly.
Source: Meilir Page-Jones, "Comparing Techniques by Means of Encapsulation and Connascence" and Meilir Page-Jones, <i>Fundamentals of Object-Oriented Design in UML</i> .	

# Object Design Activities

- An extension of analysis & evolution activities
- Expand the descriptions of partitions, layers & classes by:
  1. Adding specifications to the current model
  2. Identifying opportunities to reuse classes that already exist
  3. Restructuring the design
  4. Optimize the design
  5. Map the problem domain classes into a programming language



# Adding Specifications

- Review the current set of analysis models
  - All classes included are both sufficient and necessary to solve the problem
  - No missing attributes or methods
    - Class Student: NIM, **Identification ID**
  - No extra or unused attributes or methods
    - Class Student: NIM, ~~Lecturer Name~~
  - No missing or extra classes
- Examine the visibility of classes
  - **Private**—not visible → generally is for attributes
  - **Public**—visible to other classes → generally is for methods
  - **Protected**—visible only to members of the same super class
    - → generally, is for sub-class / child

CRC

# Access Modifiers in Java

	default	private	protected	public
Same Class	Yes	Yes	Yes	Yes
Same package subclass	Yes	No	Yes	Yes
Same package non-subclass	Yes	No	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

## Adding Specifications (cont.)

- **Decide on method signatures:**
  - Name of the method
  - Parameters or arguments to pass
  - Type of value(s) to be returned
- **Define constraints that must be preserved by the objects**
  - Preconditions, post-conditions, & invariants
  - Decide how to handle constraint violations

# Identify Opportunities for Reuse

- **Design patterns**—groupings of classes that help solve a commonly occurring problem
- **Framework**—a set of implemented classes that form the basis of an application
- **Class libraries**—also a set of implemented classes, but more general in nature than a framework
- **Components**—self-contained classes used as plug-ins to provide specific functionality
- Choice of approaches depends on the layer

# Restructuring the Design

- **Factoring**—separating aspects from a class to simplify the design
- **Normalization**—aids in identifying missing classes
- Assure all inheritance relationships support only generalization/specialization semantics

# Optimizing the Design

- Balance understandability with efficiency
- Methods:
  - Review access paths between objects
  - Review all attributes of each class
  - Review direct (number of messages sent by a method) and indirect fan-out (number of messages by methods that are induced by other methods)
  - Consider execution order of statements in often-used methods
  - Avoid re-computation by creating derived attributes and triggers
  - Consider combining classes that form a one-to-one association

# Mapping Problem-Domain Classes

- Factor out multiple inheritance if using a language that supports only single inheritance
- Factor out all inheritance if the language does not support inheritance
- Avoid implementing an object-oriented design in non-object languages

# Constraints and Contracts

- A contract is a set of constraints & guarantees

- If the requestor (client) meets the constraints, the responder (server) will guarantee certain behavior
  - Constraints must therefore be unambiguous
- Contracts document message passing between objects
- A contract is created for each visible method in a class
  - **PUBLIC**
- Should contain enough information for the programmer to understand what the method is supposed to do

- Constraint types

- Precondition—must be true *before* the method executes
- Post-condition—must be true *after* the method finishes
- Invariant—must *always* be true for all instances of a class



# Sample Contract Form

<b>Method Name:</b>	<b>Class Name:</b>	<b>ID:</b>
<b>Clients (Consumers):</b>		
<b>Associated Use Cases:</b>		
<b>Description of Responsibilities:</b>		
<b>Arguments Received:</b>		
<b>Type of Value Returned:</b>		
<b>Pre-Conditions:</b>		
<b>Post-Conditions:</b>		

# Method Specification

- Documentation details for each method
  - Allows programmers to code each method
- Must be explicit and clear
- No formal standards exist, but information should include:
  - General information (e.g., method name, class name, etc.)
  - Events—anything that triggers a method (e.g., mouse click)
  - Message passing including values passed into a method and those returned from the method
  - Algorithm specifications
  - Other applicable information (e.g., calculations, procedure calls)

# Method Specification Form



<b>Method Name:</b> insertOrder	<b>Class Name:</b> OrderList	<b>ID:</b> 100
<b>Contract ID:</b> 123	<b>Programmer:</b> J. Doe	<b>Date Due:</b> 1/1/12
<b>Programming Language:</b> <input type="checkbox"/> Visual Basic <input type="checkbox"/> Smalltalk <input type="checkbox"/> C++ <input type="checkbox"/> Java		
<b>Triggers/Events:</b> Customer places an order		
<b>Arguments Received:</b> <b>Data Type:</b>	<b>Notes:</b>	
Order	The new customer's new order.	
<b>Messages Sent &amp; Arguments Passed:</b> <b>ClassName.MethodName:</b>	<b>Data Type:</b>	<b>Notes:</b>
OrderNode.new()	Order	
OrderNode.getOrder()		
Order.getOrderNumber()		
OrderNode.setNextNode()	OrderNode	
self.middleListInsert()	OrderNode	
<b>Arguments Returned:</b> <b>Data Type:</b>	<b>Notes:</b>	
void		
<b>Algorithm Specification:</b> See Figures 8-30 and 8-31.		
<b>Misc. Notes:</b> None.		

# Summary

## 1. Basic Characteristics of Object Orientation (review)

- Class, Polymorphism, and Inheritance

## 2. Design Criteria—coupling, cohesion & connascence

1. Low Coupling 👍
2. High Cohesion: Method and Class 👍
3. Avoid Connascence for a better program 👍

## 3. Object Design Activities

- Review all of design: reuse, complete our class, and remove unnecessary attributes, check the access modifiers

## 4. Constraints and Contracts

- Build a contract for every method in a class

## 5. Method Specification

- We can see the relationship between method to every classes
- Define the detail of our method: showing the client (receiver of the messages)

## References

Denis, Wixom, Tegarden. (2015). Systems Analysis and Design: An Object-Oriented Approach with UML. 5<sup>th</sup> edition. ISBN: 978-1-118-80467-4, John Wiley & Sons, Inc, Denver (USA)