



# Software Engineering I

Dr. Elham Mahmoudzadeh  
Isfahan University of Technology  
[mahmoudzadeh@iut.ac.ir](mailto:mahmoudzadeh@iut.ac.ir)

2021

The background features a light gray gradient with several realistic water droplets of varying sizes scattered across the surface. In the center, there is a faint, circular logo. The logo consists of a gear-like outer ring with Persian text 'دانشگاه صنعتی اصفهان' (University of Technology, Isfahan) written along its top arc. Inside the gear is a stylized sunburst or star-like emblem.

# **Chapter 8**

## **Class and Method design**

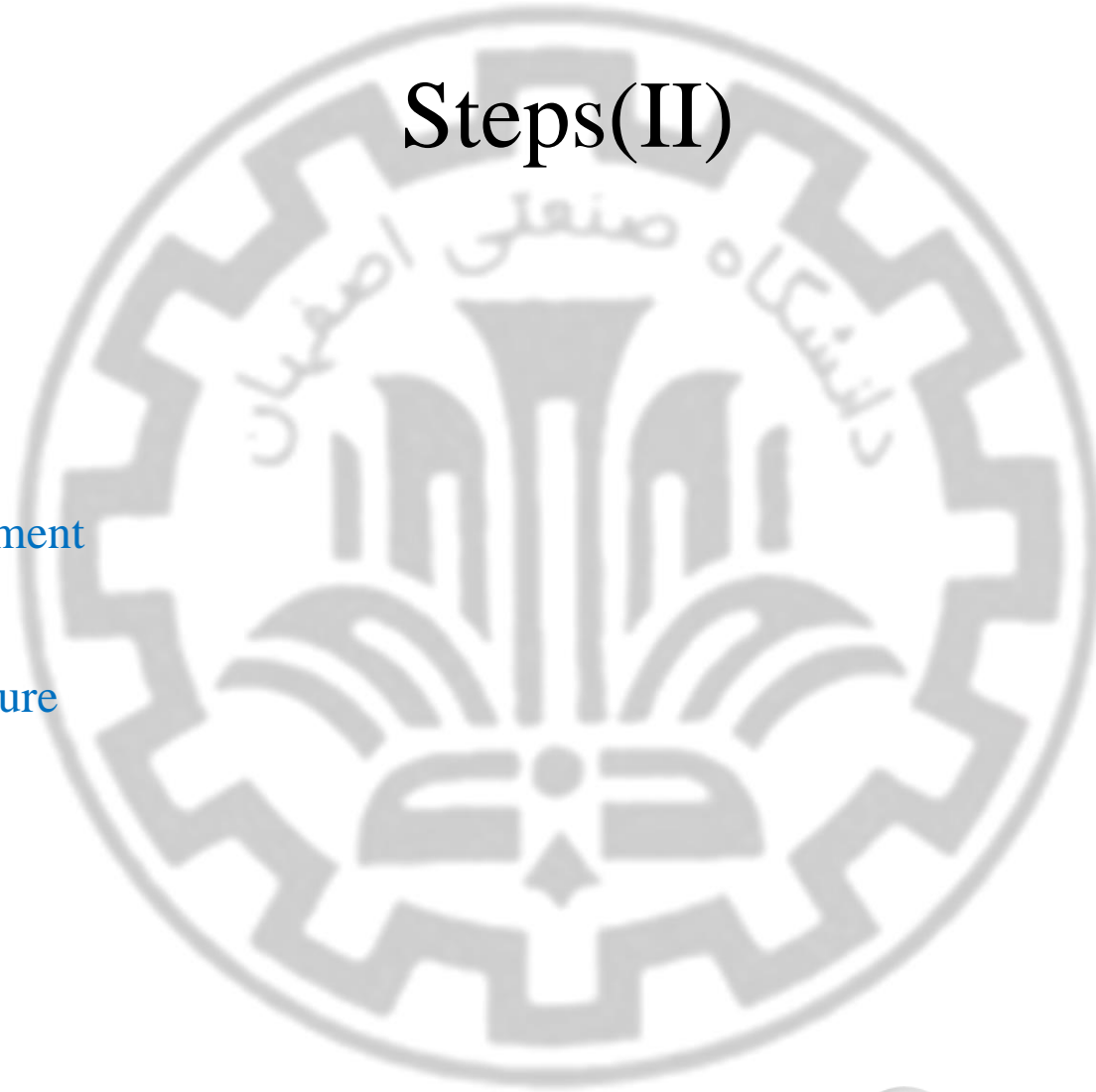
# Steps(I)

1. Preparing proposal
2. Requirements determination
  - User story
3. Abstract Business Process Modelling
4. Analysis
  - Functional Modelling
  - Structural Modelling
  - Behavioral Modelling

## Steps(II)

### 5. Design

- Optimization
- Database Management
- User Interface
- Physical Architecture



# Design Criteria

A good design is one that balances trade-off s to minimize the total cost of the system over its entire lifetime.

- **Coupling**
- **Cohesion**




# 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 sent 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 are based on material from Page-Jones, *The Practical Guide to Structured Systems*; Myers, *Composite/Structured Design*; Edward Yourdon and Larry L. Constantine, *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design* (Englewood Cliffs, NJ: Prentice-Hall, 1979).

# 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) has 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.
	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.
Worse		

Based upon material from Page-Jones, *Fundamentals of Object-Oriented Design in UML*.

# Types of interaction coupling

Level	Type	Description
Good ↓ Bad	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.
	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 are based on material from Meilir Page-Jones, *The Practical Guide to Structured Systems Design*, 2nd Ed. (Englewood Cliffs, NJ: Yardon Press, 1988); Glenford Myers, *Composite/Structured Design* (New York: Van Nostrand Reinhold, 1978).



# SOLID Principles

- Single Responsibility (SRP)
- Open-Close principle (OCP)
- Liskov Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)

# Open Closed Principle (OCP)

- Classes should be open for extension but closed for modification.
- OCP states that we should be able to add new features to our system without having to modify our set of preexisting classes.
- Reduce the coupling between classes to the abstract level.
- Instead of creating relationships between two concrete classes, we create relationships between a concrete class and an abstract class or an interface.
- Interface is a shared boundary across which two or more separate components of a computer system exchange information.

# Liskov Substitution Principle (LSP)

- Subclasses should be substitutable for their base classes.
- Subclasses should not limit their base classes, for example should not change “public” variable to “private”.

# Interface Segregation Principle (ISP)

- Many specific interfaces are better than a single, general interface.
- Any interface we define should be highly cohesive.

# Dependency Inversion Principle (DIP)

- The structure of many older software architectures is hierarchical. At the top of the architecture, “control” components rely on lower-level “worker” components to perform various cohesive tasks.
- Consider a simple program with three components. The intent of the program is to read keyboard strokes and then print the result to a printer. A control module, *C*, coordinates two other modules—a keystroke reader module, *R*, and a module that writes to a printer, *W*.
- The design of the program is coupled because *C* is highly dependent on *R* and *W*. To remove the level of dependence that exists, the “worker” modules *R* and *W* should be invoked from the control module *S* using abstractions.



# Dependency Inversion Principle (Cnt'd)

- In object-oriented software engineering, abstractions are implemented as abstract classes, **R\*** and **W\***. These abstract classes could then be used to invoke worker classes that perform any read and write function.
- Therefore class, **C**, invokes abstract classes, **R\*** and **W\***, and the abstract class points to the appropriate worker-class (e.g., the **R\*** class might point to a *read()* operation within a **keyboard** class in one context and a *read()* operation within a **sensor** class in another. This approach reduces coupling and improves the testability of a design.

# Dependency Inversion Principle (Cnt'd)

- Depend upon abstractions. Do not depend upon concretions.
- Formalizes the concept of abstract coupling and clearly states that we should couple at the abstract level, not at the concrete level.

*High-level modules (classes) should not depend [directly] upon low-level modules. Both should depend on abstractions.*

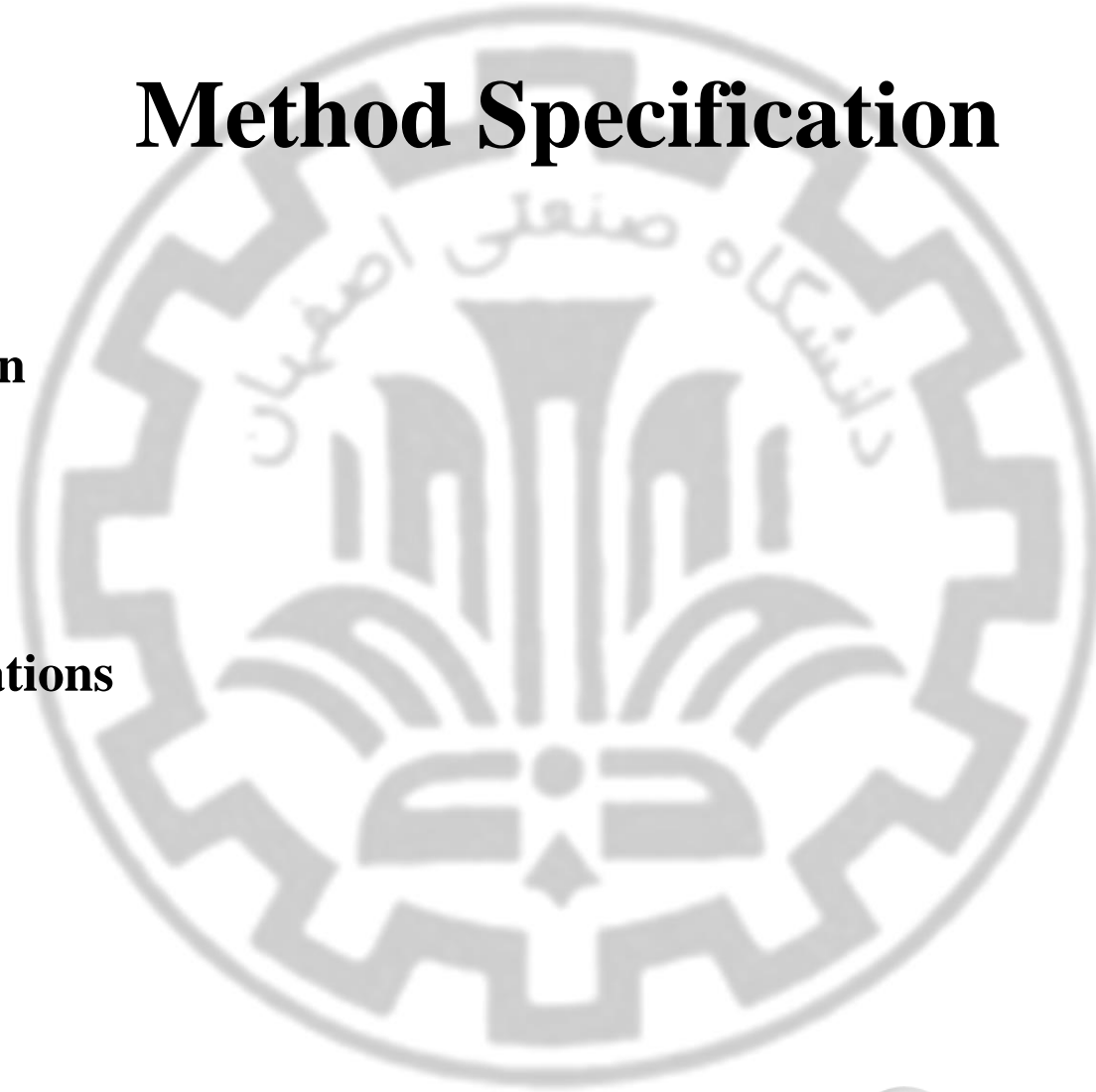
*Abstractions should not depend on details. Details should depend on abstractions.*

# Adding Specifications for Object Design

- There are no missing attributes or methods and no extra or unused attributes or methods in each class.
- Finalize the visibility (hidden or visible) of the attributes and methods in each class.
- Decide on the signature of every method in every class. The *signature* of a method comprises three parts: the name of the method, the parameters or arguments that must be passed to the method, and the type of value that the method will return to the calling method.
- Any constraints that must be preserved by the objects.

# Method Specification

- **General Information**
- **Events**
- **Message Passing**
- **Algorithm Specifications**



# Reference

- **Dennis, Wixon, Tegarden**, “System Analysis and Design, An Object Oriented Approach with UML”, 5th Edition, 2015.
- **R. G. Pressman, B. R. Maxim**, Software Engineering\_ A Practitioner’s Approach, 8th Edition, 2014.