UNIVERSITAS INDONESIA
Veritas, Probitas, Iustitia

FAKULTAS ILMU KOMPUTER

Module 03: OO Principles & Software Maintainability

Advanced Programming

Baginda Anggun Nan Cenka
Editor: Teaching Team (Lecturer and Teaching Assistant)
Advanced Programming
Semester Genap 2024
Fasilkom UI

Author: *Baginda Anggun Nan Cenka*
Email: nancenka@cs.ui.ac.id
© 2024 Fakultas Ilmu Komputer Universitas Indonesia
This work uses license: Creative Commons Attribution-ShareAlike 4.0 International (CC BY-SA 4.0)

Table of Contents

Table of Contents 1
Learning Objectives 3
References 3
1. Object-Oriented Principles 4
    Object-Oriented Programming 5
    Key Concept of OOP 6
    Class 7
    Object 8
    Abstraction 9
    Encapsulation 11
    Inheritance 13
    Polymorphism 15
    OO Principles 17
    Brief History of SOLID Principle 20
    Overview of SOLID Principle 21
    Importance of SOLID Principles 22
    Single Responsibility Principle (SRP) 23
    Open-Closed Principle (OCP) 26
    Liskov Substitution Principle (LSP) 29
    Interface Segregation Principle (ISP) 33
    Dependency Inversions Principle (DIP) 39
    Conclusion about SOLID Principles 43
2. Software Maintainability 43
    Software Development Life Cycle 45
    What is Maintainability? 46
    Why is Maintainability Important? 49
    How to Measure Software Maintainability? 57
    How to Achieve Maintainability? 60
    Misunderstanding about Maintainability 91
    Conclusion About Software Maintainability 94
3. Tutorial & Exercise 96
    Development 96
        Run Application 106
        Exercise 106
        Reflection 107
    Grading Scheme 108
        Scale 108
        Components 108
        Rubrics 108

Learning Objectives

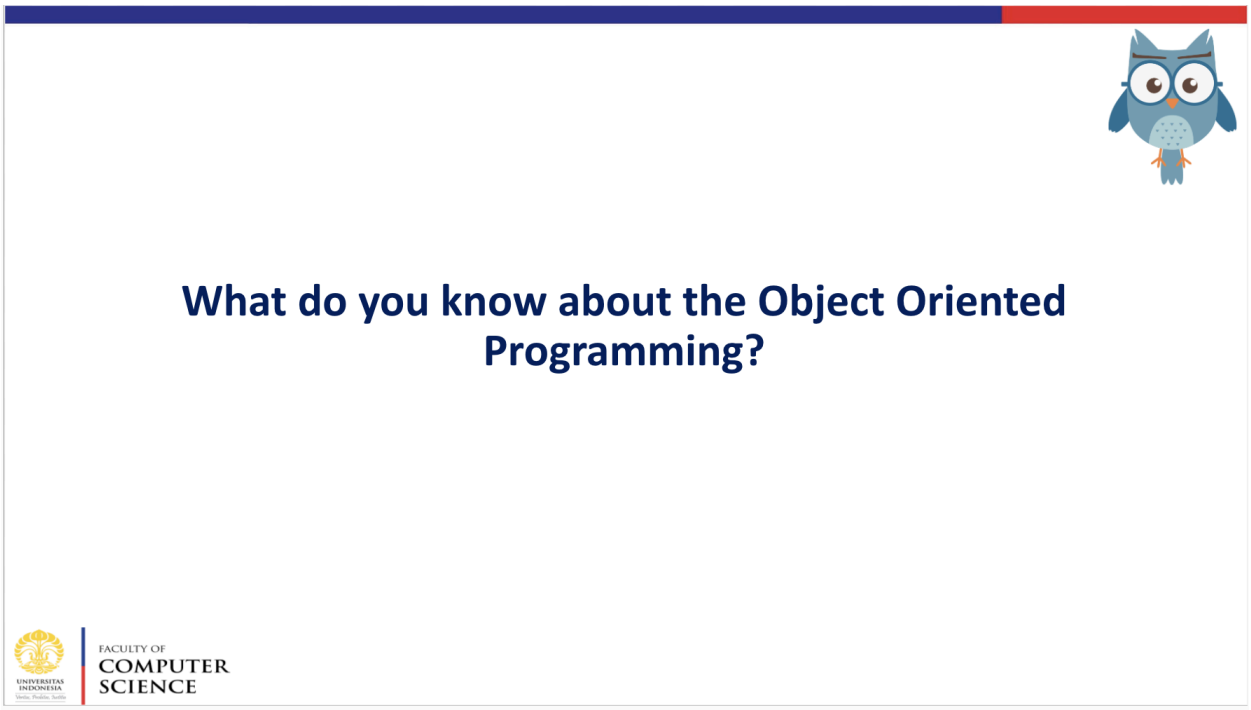
1. Students can understand and apply object-oriented principles
2. Students can understand and apply guidelines for developing maintainable software.

References

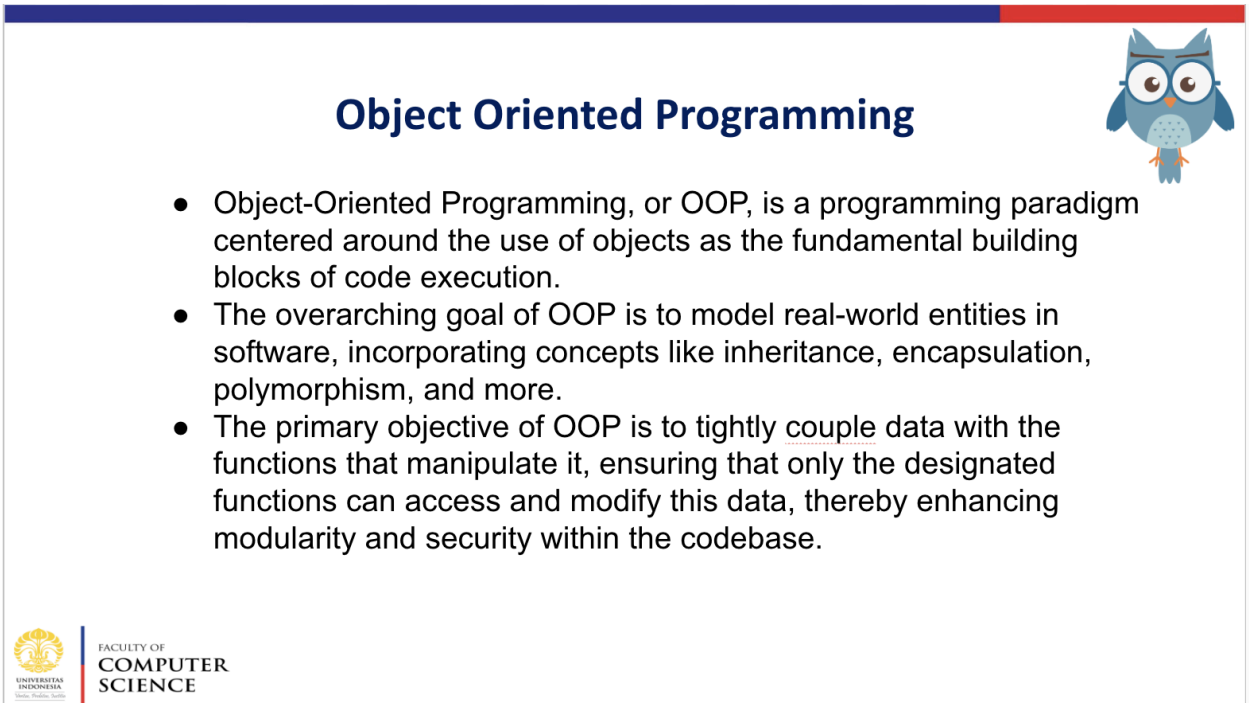
1. Martin, R. C. (2017). *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Boston, MA: Prentice Hall. ISBN: 978-0-13-449416-6

2. Visser, J., & Rigal, S. (2016). *Building Maintainable Software, Java Edition: Ten Guidelines for Future-Proof Code*. O'Reilly Media.

1. Object-Oriented Principles



Before discussing object-oriented principles, please explain what you know about object-oriented programming.

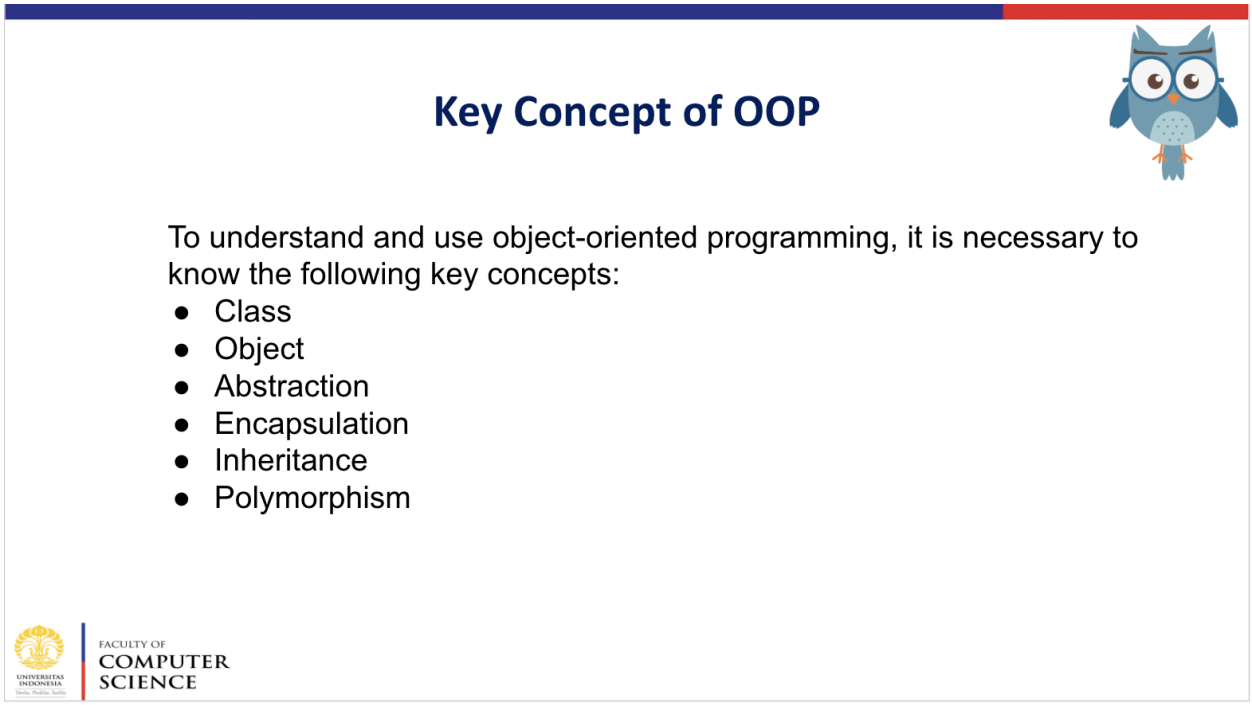


Object-oriented programming, or OOP, is a programming paradigm centered around the use of objects as the fundamental building blocks of code execution. In OOP, objects encapsulate both data and methods.

These objects, visible and interactive to users, execute tasks as directed by the programmer. The overarching goal of OOP is to model real-world entities in software, incorporating concepts like inheritance, encapsulation, polymorphism, and more.

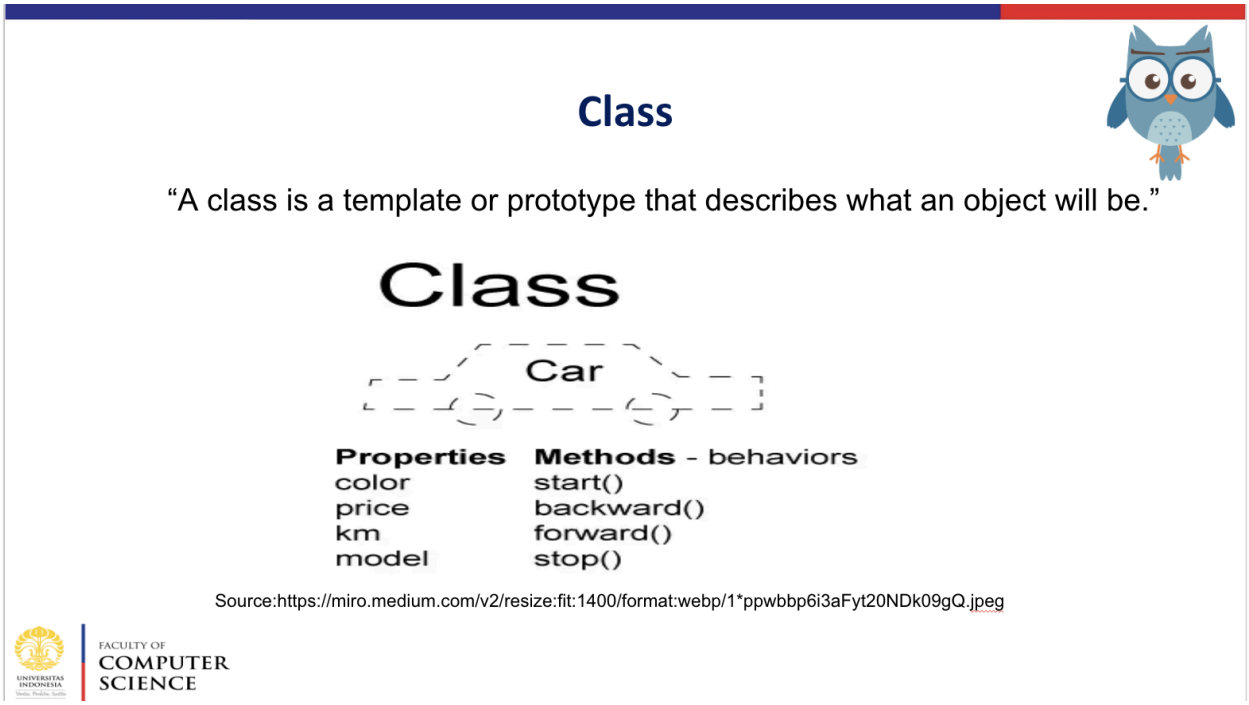
The primary objective of OOP is to tightly couple data with the functions that manipulate it, ensuring that only the designated functions can access and modify this data, thereby enhancing modularity and security within the codebase.

Key Concept of OOP



Object-oriented programming (OOP) is a programming paradigm that revolves around the concept of "objects", which are instances of "classes". Here's a breakdown of the key concepts:
1. Class: A class is a blueprint or template for creating objects. It defines the properties (attributes) and behaviors (methods) that all objects of that class will have.
2. Object: An object is an instance of a class. It is a concrete entity created based on the blueprint provided by the class.
3. Abstraction: Abstraction is the concept of hiding the complex implementation details and showing only the necessary features of an object.
4. Encapsulation: Encapsulation refers to the bundling of data (attributes) and methods that operate on that data into a single unit, known as a class.
5. Inheritance: Inheritance is a mechanism in OOP that allows a new class (called a subclass or derived class) to inherit properties and methods from an existing class (called a superclass or base class).
6. Polymorphism: Polymorphism allows objects of different classes to be treated as objects of a common superclass.

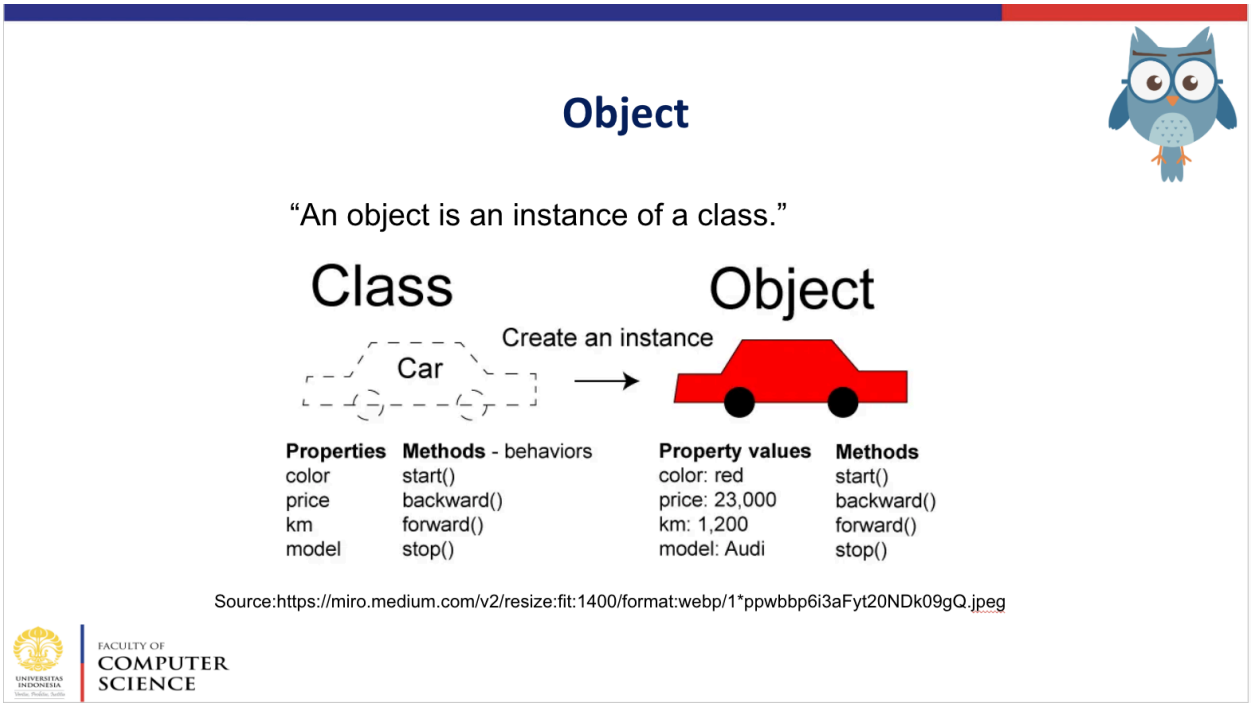
Class



Classes serve as the foundational blueprint for creating objects, acting as templates that define both their attributes (data) and behaviors (methods). Typically, a class comprises member fields, member methods, and a special constructor method. By utilizing classes, you can instantiate multiple objects with shared behaviors, eliminating the need to duplicate code. This is particularly useful for objects recurring throughout your codebase. In Java, everything is associated with classes and objects, each possessing attributes and methods. For instance, in practical terms, consider a car as an object. A car would possess attributes like weight and color, alongside methods such as drive and brake. It's essential to design a class before instantiating an object to establish its structure and functionality.

After understanding the explanation, please write in your own words, what is Class?

Object

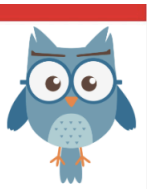


An object serves as a fundamental element in Object-Oriented Programming, symbolizing real-world entities. It's essentially an instance of a class. When we instantiate an object, we're essentially bringing it to real-world entities like cars, bicycles, or dogs, each with their unique attributes and behaviors. In a typical Java program, numerous objects are created, interacting by invoking methods.

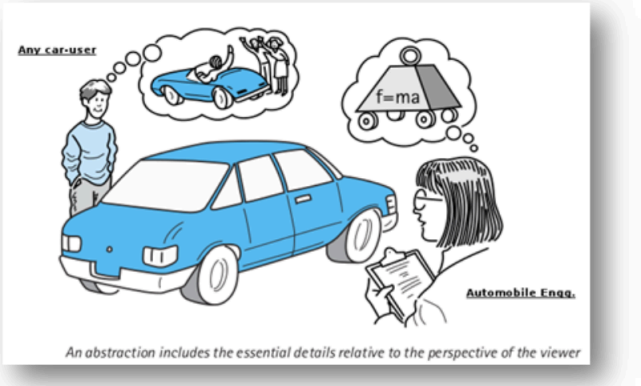
To instantiate an object of the Main class in Java, specify the class name, followed by the object name, and utilize the keyword "new".

After understanding the explanation, please write in your own words, what is Object?

Abstraction



Process of hiding certain details and showing only essential information to the user.



Source:https://reachmnadeem.files.wordpress.com/2014/05/booc1.png

FACULTY OF COMPUTER SCIENCE

Abstraction entails concealing the intricacies of implementation while presenting simplified interfaces. Data Abstraction, specifically, involves showcasing only essential details to the user, excluding trivial or non-essential elements. It can be described as the process of identifying and displaying only the necessary characteristics of an object, disregarding irrelevant details. These characteristics and behaviors not only distinguish an object from others of the same type but also aid in categorizing or grouping objects.

Consider a practical scenario where a person drives a car. The driver understands that pressing the accelerator increases the car's speed and applying brakes halts it, yet lacks knowledge of the internal workings of these mechanisms. This illustrates abstraction – focusing on the functional outcomes rather than the underlying mechanisms.

In Java, abstraction is achieved through interfaces and abstract classes. An abstract class, declared with the "abstract" keyword, can contain both abstract methods (without implementation) and non-abstract methods (with implementation). Abstraction enables developers to concentrate on an object's actions rather than their implementation details.

Abstraction

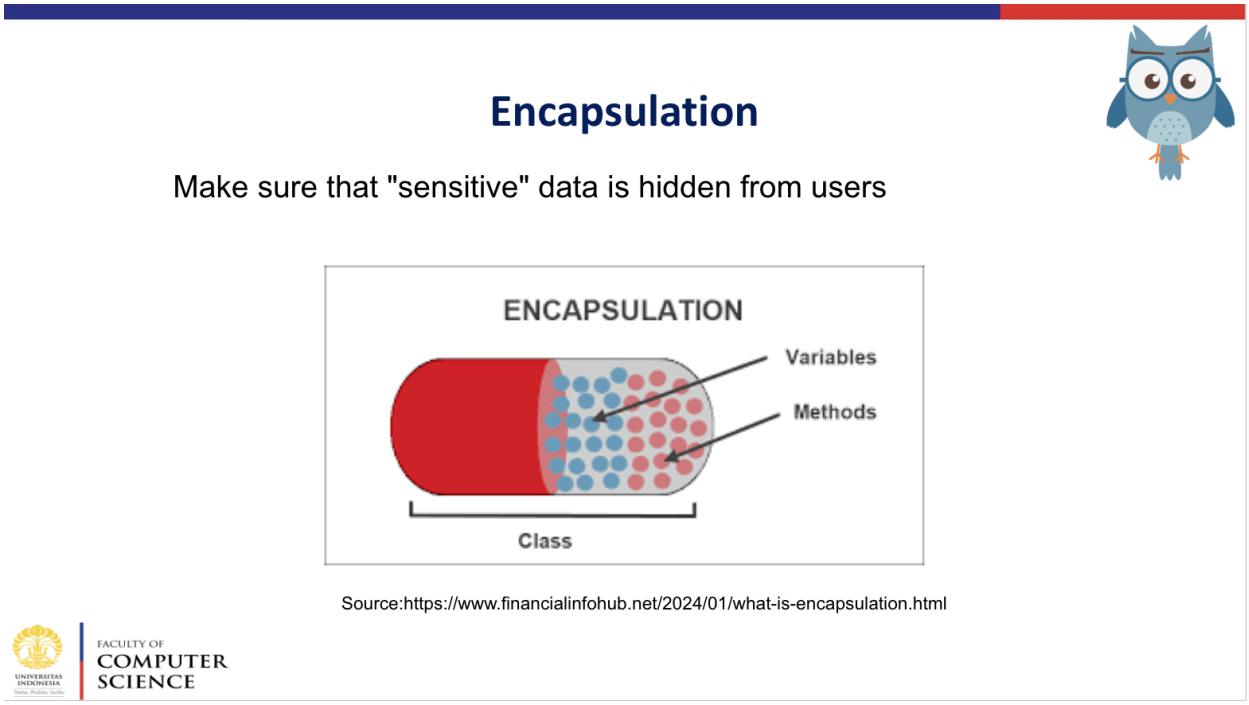
```
abstract class Animal {
  // Abstract method (does not have a body)
  public abstract void animalSound();
  // Regular method
  public void sleep() {
    System.out.println("Zzz");
  }
}

// Subclass (inherit from Animal)
class Cat extends Animal {
  public void animalSound() {
    // The body of animalSound() is provided here
    System.out.println("The cat says: ...");
  }
}

class Main {
  public static void main(String[] args) {
    Pig myCat = new Cat(); // Create a Cat object
    myCat.animalSound();
    myCat.sleep();
  }
}
```

FACULTY OF COMPUTER SCIENCE

To understand Abstraction, you can take a look at the code snippet above.

After understanding the explanation, please write in your own words, what is Abstraction?

Encapsulation



Encapsulation in Java involves bundling code and data into a unified entity. It entails concealing the internal state or representation of an object from external access and providing publicly accessible methods for interaction. This approach enables the concealment of specific details while regulating access to the object's internal workings, akin to a capsule containing various medications.

From a technical standpoint, encapsulation ensures that variables or data within a class remain inaccessible to other classes, accessible solely through member functions of the declaring class.

Achieving encapsulation typically involves declaring class variables as private and providing public methods within the class for setting and retrieving variable values. Encapsulated classes facilitate easier testing, making them well-suited for unit testing purposes.

Encapsulation

```
public class Person {
  private String name; // private = restricted access

  // Getter
  public String getName() {
    return name;
  }

  // Setter
  public void setName(String newName) {
    this.name = newName;
  }
}
```

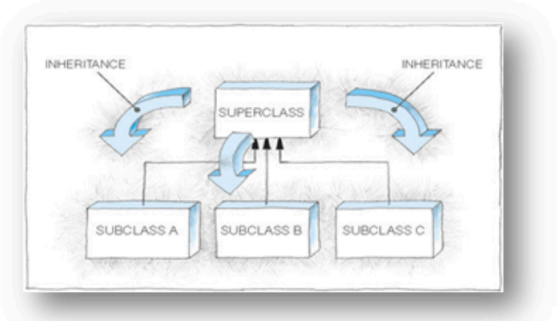
FACULTY OF COMPUTER SCIENCE

To understand Encapsulation, you can take a look at the code snippet above.

After understanding the explanation, please write in your own words, what is Encapsulation?

Inheritance

"IS-A" relationship, also termed as a parent-child relationship.



Source:https://reachmnadeem.files.wordpress.com/2014/05/booc14.png

FACULTY OF COMPUTER SCIENCE

Inheritance stands as a fundamental concept in Object-Oriented Programming (OOP), serving as a key mechanism in Java where one class inherits the attributes (fields and methods) of another. This inheritance is facilitated using the "extends" keyword and reflects an "IS-A" relationship, also termed as a parent-child relationship.

The underlying principle of inheritance in Java is the ability to construct new classes based on existing ones. By inheriting from a parent class, you gain access to its methods and fields, thus enabling their reuse. Additionally, you have the flexibility to introduce new methods and fields within the inheriting class.

Here are some frequently used terminologies related to inheritance:

- Superclass: This refers to the class whose attributes are inherited, also known as the base or parent class.
- Subclass: Conversely, the class inheriting from another is termed the subclass, also referred to as the derived, extended, or child class. The subclass can augment the features of the superclass with its fields and methods.

The utility of inheritance in Java lies in its facilitation of method overriding, enabling runtime polymorphism, and fostering code reusability.

Inheritance

```
class Vehicle {
  protected String brand = "Ford";        // Vehicle attribute
  public void honk() {                    // Vehicle method
    System.out.println("Tuut, tuut!");
  }
}

class Car extends Vehicle {
  private String modelName = "Mustang";    // Car attribute
  public static void main(String[] args) {

    // Create a myCar object
    Car myCar = new Car();

    // Call the honk() method (from the Vehicle class) on the myCar object
    myCar.honk();

    // Display the value of the brand attribute (from the Vehicle class) and the value of the modelName from the Car class
    System.out.println(myCar.brand + " " + myCar.modelName);
  }
}
```

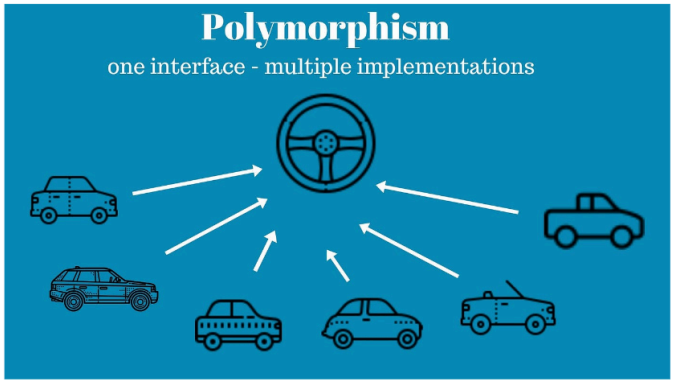
FACULTY OF COMPUTER SCIENCE

To understand Inheritance, you can take a look at the code snippet above

After understanding the explanation, please write in your own words, what is Inheritance?

Polymorphism

‘Multiple bodies’ that provide the same behavior.



Source:https://viktor-kukurba.medium.com/object-oriented-programming-in-javascript-3-polymorphism-fb564c9f1ce8

Polymorphism in Object-Oriented Programming (OOP) denotes the capability of a language to handle data diversely based on input types, allowing a single action to be executed in various ways. The term "polymorphism" originates from the Greek words "poly" meaning many and "morphs" meaning forms, collectively conveying the notion of "many forms."

In the context of OOP, polymorphism also encompasses the efficient distinction between entities sharing the same name. It embodies the concept of an entity's ability to manifest in multiple forms.

For instance, consider the task of drawing, which can involve diverse shapes such as triangles, rectangles, and circles. In Java, polymorphism can be exemplified through a method bearing the same name yet possessing different method signatures and functionalities.

Polymorphism

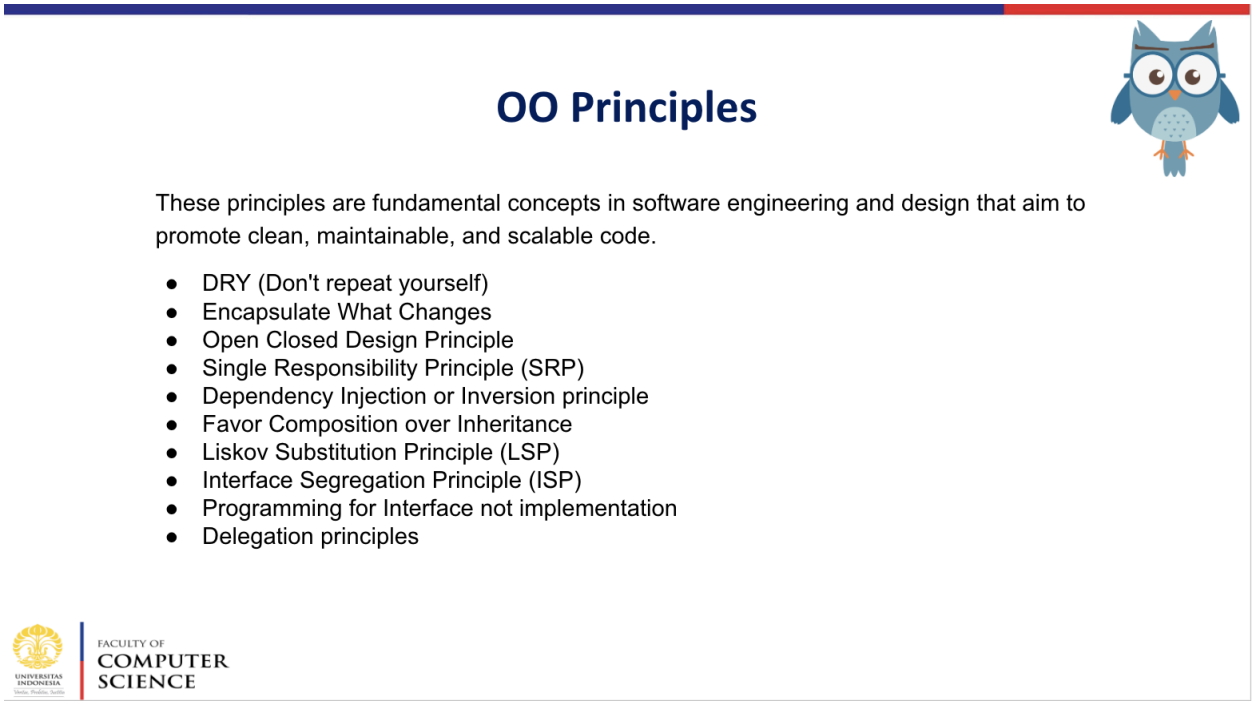
```
class Animal {
  public void animalSound() {
    System.out.println("The animal makes a sound");
  }
}

class Cat extends Animal {
  public void animalSound() {
    System.out.println("The cat says: …");
  }
}

class Dog extends Animal {
  public void animalSound() {
    System.out.println("The dog says: bow wow");
  }
}
```

FACULTY OF COMPUTER SCIENCE

To understand Polymorphism, you can take a look at the code snippet above

After understanding the explanation, please write in your own words, what is Polymorphism?

OO Principles



These principles are fundamental concepts in software engineering and design that aim to promote clean, maintainable, and scalable code. Let's break down each one:

1. DRY (Don't Repeat Yourself): This principle suggests that duplication in logic should be avoided. Instead, reusable code should be abstracted into functions, methods, or modules. The idea is to have a single, authoritative source of truth for each piece of knowledge within a system.

2. Encapsulate What Changes: This principle advises encapsulating the parts of a system that are likely to change in the future. By encapsulating such changes, you can minimize the impact of modifications on other parts of the system, thus improving maintainability and flexibility.

3. Open-Closed Design Principle: This principle states that software entities (classes, modules, functions, etc.) should be open for extension but closed for modification. In other words, you should be able to extend the behavior of a system without modifying its existing codebase.

4. Single Responsibility Principle (SRP): This principle suggests that a class or module should have only one reason to change. In other words, it should have only one responsibility or job. This improves readability, maintainability, and testability by making classes or modules more focused.

5. Dependency Injection or Inversion Principle: This principle states that high-level modules should not depend on low-level modules, but rather both should depend on abstractions. Dependency Injection (DI) is a technique that implements this principle by injecting dependencies into a class from the outside rather than creating them internally.

6. Favor Composition over Inheritance: This principle advises preferring object composition over class inheritance when building software components. Composition allows for more flexibility and is less prone to the issues that can arise from deep inheritance hierarchies.

7. Liskov Substitution Principle (LSP): This principle states that objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program. In other words, subclasses should be substitutable for their base classes without altering the desirable properties of the program.

8. Interface Segregation Principle (ISP): This principle suggests that clients should not be forced to depend on interfaces they do not use. Instead, interfaces should be segregated into smaller, more specific ones, tailored to the needs of the clients.

9. Programming for Interface not Implementation**: This principle encourages coding to interfaces rather than concrete implementations. By programming to interfaces, you decouple your code from specific implementations, making it more flexible and easier to maintain.

10. Delegation Principles: Delegation is a design pattern where an object forwards method calls to a delegate object. This allows for modular, reusable code and promotes the Single Responsibility Principle by separating concerns.

These principles collectively contribute to the creation of software that is modular, maintainable, and extensible, ultimately leading to more robust and scalable systems.

What do you know about the S.O.L.I.D principle?

FACULTY OF COMPUTER SCIENCE

Explain what you know about SOLID Principles first!

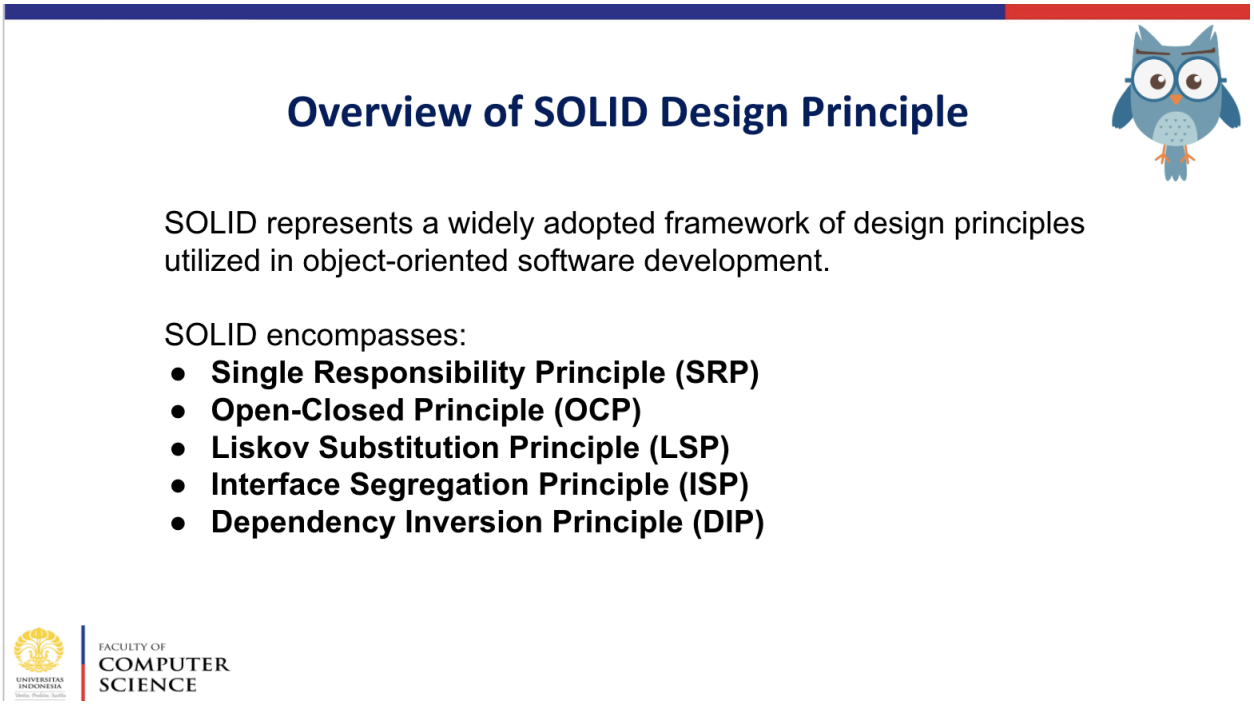
Brief History of SOLID Design Principle

- The SOLID principles, introduced by **Robert C. Martin in his 2000** essay "Design Principles and Design Patterns," were later coined with the **acronym by Michael Feathers**.
- Martin **emphasized** the inevitability of software evolution and complexity, cautioning that without proper design principles, software can become inflexible, fragile, stagnant, and difficult to maintain.
- The SOLID principles were specifically devised t**o address and alleviate these problematic design patterns**.

FACULTY OF COMPUTER SCIENCE

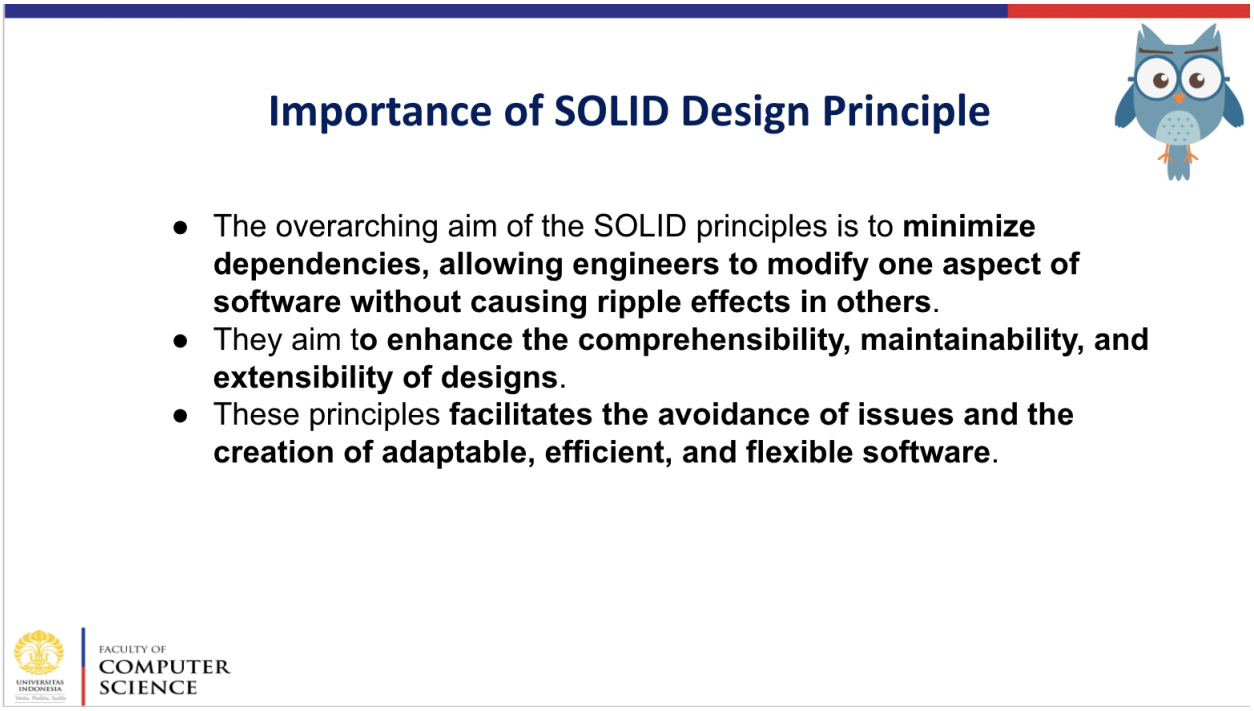
The SOLID principles, introduced by Robert C. Martin in his 2000 essay "Design Principles and Design Patterns," were later coined with the acronym by Michael Feathers. Martin outlined in his paper the issues of software decay and proposed five object-oriented class design principles as remedies. It's important to note that while Martin didn't initially use the SOLID acronym in his paper, Feathers introduced it in 2004. Martin emphasized the inevitability of software evolution and complexity, cautioning that without proper design principles, software can become inflexible, fragile, stagnant, and difficult to maintain. The SOLID principles were specifically devised to address and alleviate these problematic design patterns.

Overview of SOLID Principle



SOLID represents a widely adopted framework of design principles utilized in object-oriented software development. Comprising five fundamental principles, SOLID encompasses the single responsibility principle, open-closed principle, Liskov substitution principle, interface segregation principle, and dependency inversion principle.

These principles collectively aid software developers in crafting resilient, verifiable, adaptable, and manageable object-oriented software architectures. Each principle addresses specific challenges encountered during software development, contributing to the overall robustness of the system. Software engineers routinely employ all five principles, recognizing their significant advantages in facilitating development processes.



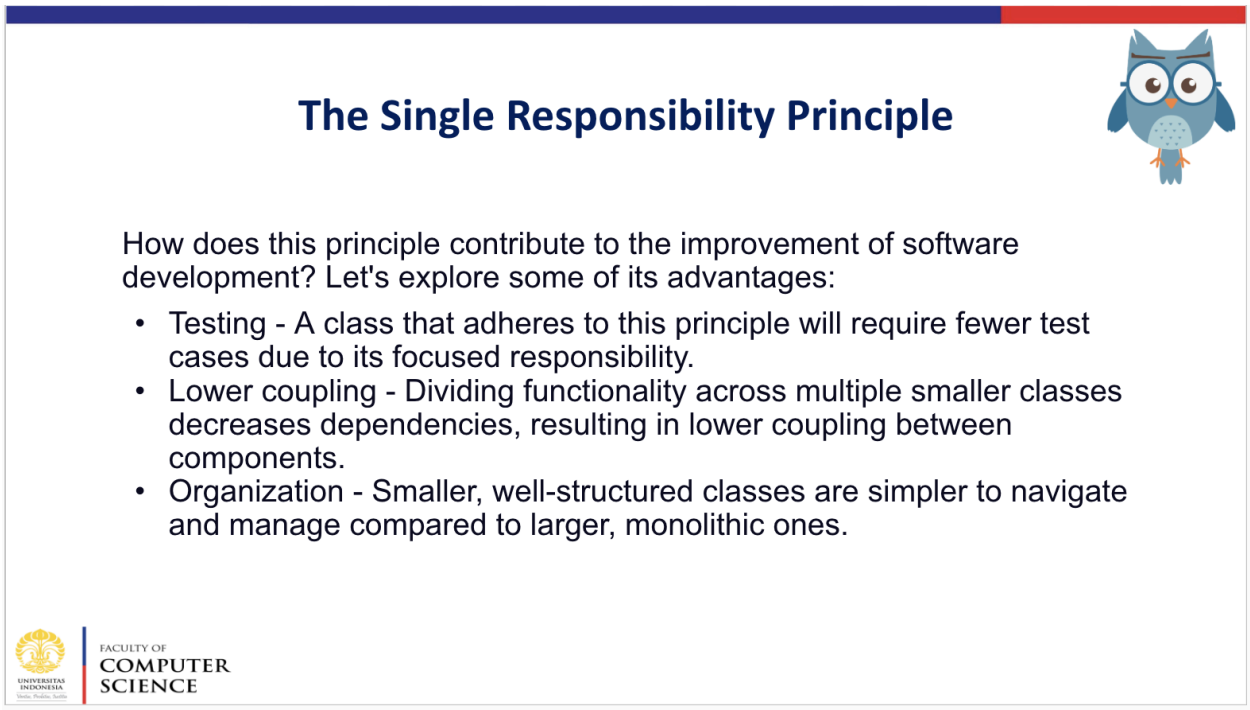
The overarching aim of the SOLID principles is to minimize dependencies, allowing engineers to modify one aspect of software without causing ripple effects in others. Moreover, they aim to enhance the comprehensibility, maintainability, and extensibility of designs. Ultimately, adherence to these principles facilitates the avoidance of issues and the creation of adaptable, efficient, and flexible software. While implementing these principles offers numerous advantages, it often results in the production of lengthier and more intricate code. Consequently, this may prolong the design phase and add complexity to development. Nevertheless, the additional time and effort invested are justified by the considerable benefits of easier maintenance, testing, and expansion of the software.

Single Responsibility Principle (SRP)

Let's start with the single responsibility principle, which asserts that a class should have a singular responsibility. For instance, in software development, different team members handle various tasks: front-end designers handle design, testers perform testing, and backend developers manage backend development. Each person has a distinct role or responsibility.

Often, programmers mistakenly add new features or behaviors directly into existing classes, resulting in lengthy, complex code that becomes cumbersome to modify later. To address this, it's advisable to organize your application into layers and decompose large classes into smaller ones or modules.

The single responsibility principle dictates that each Java class should execute only one function. Combining multiple functionalities within a single class complicates the code and makes it challenging to maintain. Adhering to this principle ensures code clarity and facilitates easier maintenance.



How does this principle contribute to the improvement of software development? Let's explore some of its advantages:

- Testing - A class that adheres to this principle will require fewer test cases due to its focused responsibility.
- Lower coupling - Dividing functionality across multiple smaller classes decreases dependencies, resulting in lower coupling between components.
- Organization - Smaller, well-structured classes are simpler to navigate and manage compared to larger, monolithic ones.

The Single Responsibility Principle

Student.java

```
public class Student{
      public void printDetails() {
            //functionality of the method
      }

      pubic void calculatePercentage(){
            //functionality of the method
      }

      public void addStudent(){
            //functionality of the method
      }
}
```

Violation

FACULTY OF COMPUTER SCIENCE

Imagine we have a class called Student, which currently contains three methods: printDetails(), calculatePercentage(), and addStudent(). Consequently, the Student class is responsible for printing student details, calculating percentages, and managing the database. Applying the single responsibility principle suggests dividing these responsibilities into three distinct classes, each dedicated to fulfilling a single purpose.

The Single Responsibility Principle

Student.java

```
public class Student{
      public void addStudent(){
            //functionality of the method
      }
}
```

PrintStudentDetails.java

```
public class PrintStudentDetails{
      public void printDetails(){
            //functionality of the method
      }
}
```

Solution

Percentage.java

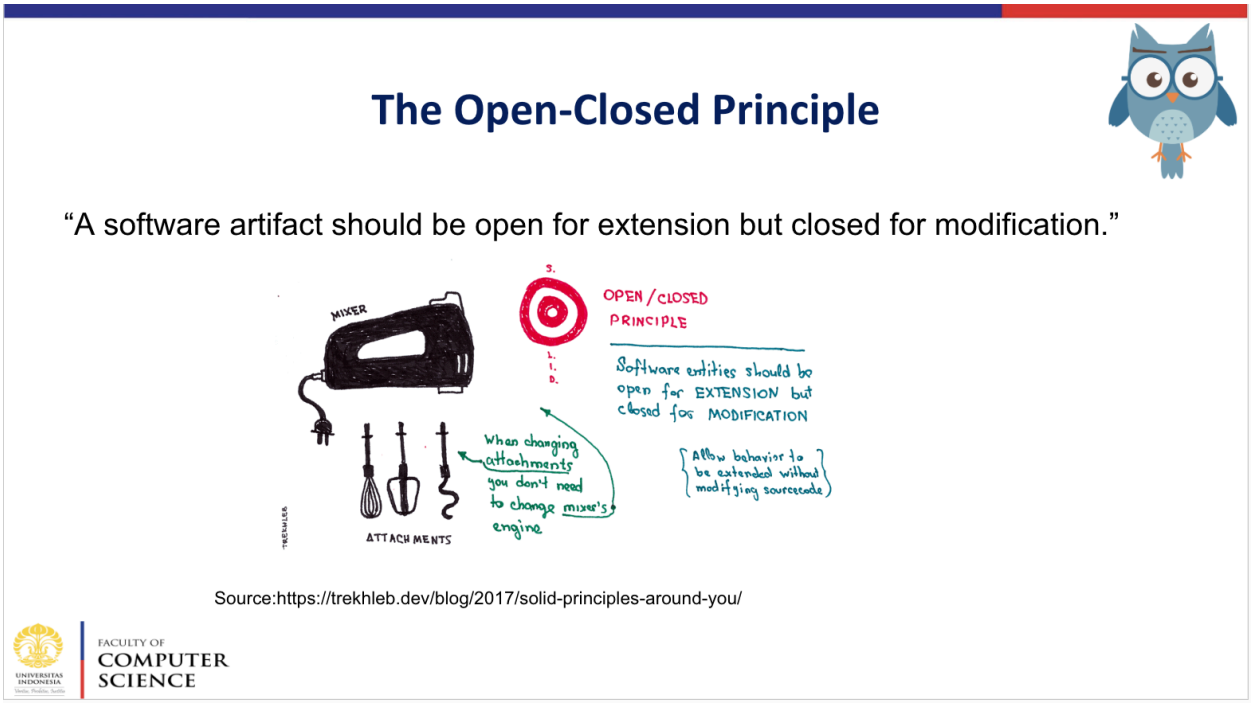
```
public class Percentage{
      public void calculatePercentage(){
            //functionality of the method
      }
}
```

FACULTY OF COMPUTER SCIENCE

The previous code snippet violates the single responsibility principle. To achieve the goal of the principle, we should implement a separate class that performs a single functionality only.

After understanding the explanation, please write in your own words, what is SRP?

Open-Closed Principle (OCP)



Now, let's delve into the "O" in SOLID, which stands for the open-closed principle. Coined by Bertrand Meyer in 1988, the Open-Closed Principle (OCP) dictates that a software artifact should be open for extension but closed for modification. Put simply, the behavior of a software artifact should be extendable without necessitating changes to the artifact itself. Adhering to

this principle helps prevent the introduction of new bugs into a functioning application when making modifications to existing code, except in cases where bugs need fixing.

According to this principle, "software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification," meaning that it should be possible to extend the behavior of a class without directly altering its code. For example, if developer A releases an update for a library or framework, developer B can extend the existing class created by developer A to make modifications or add new features, rather than directly modifying the class.

The Open-Closed Principle

VehicleInfo.java

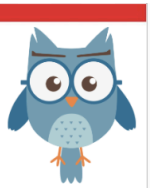
```
public class VehicleInfo{

    public double vehicleNumber(Vehicle vcl){

        if (vcl instanceof Car){
            return vcl.getNumber();
        }

        if (vcl instanceof Bike){
            return vcl.getNumber();
        }
    }
}
```

Violation

FACULTY OF COMPUTER SCIENCE

The open-closed principle maintains that a module should be amenable to extension to accommodate new requirements while remaining impervious to direct modification. Extending the module permits the integration of fresh functionalities. Let's illustrate this principle with an example. Suppose, VehicleInfo is a class and it has the method vehicleNumber() that returns the vehicle number.

The Open-Closed Principle



VehicleInfo.java

```
public class VehicleInfo{

      public double vehicleNumber() {
            //functionality
      }
}

public class Car extends VehicleInfo{

      public double vehicleNumber(){
            return this.getValue();
      }
}

public class Car extends Truck{
      public double vehicleNumber(){
            return this.getValue();
      }
}
```

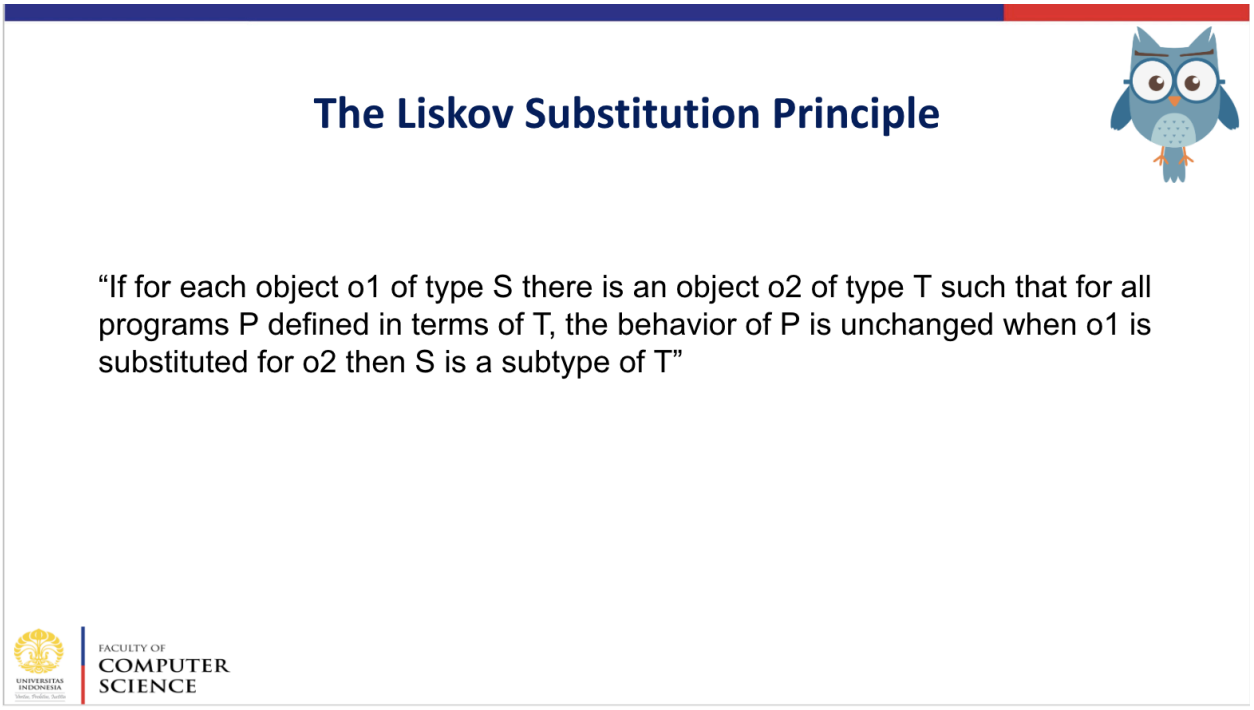
Solution

FACULTY OF COMPUTER SCIENCE

If we want to add another subclass named Truck, simply, we add one more if statement that violates the open-closed principle. The only way to add the subclass and achieve the goal of principle by overriding the vehicleNumber() method, as we have shown below.

After understanding the explanation, please write in your own words, what is OCP?

Liskov Substitution Principle (LSP)

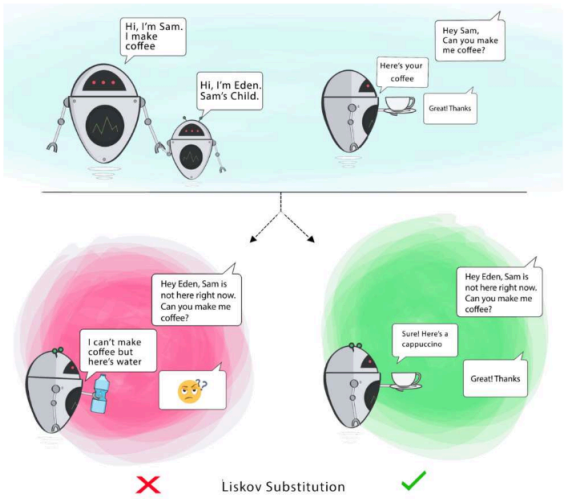


Moving forward, let's discuss the Liskov substitution principle, which is often considered the most intricate among the five principles. Introduced by Barbara Liskov in 1987, this principle states that "Derived or child classes must be substitutable for their base or parent classes." Essentially, it ensures that any child class can be used interchangeably with its parent class without causing unexpected behavior. In simpler terms, if class A is a subtype of class B, we should be able to replace B with A seamlessly, maintaining the program's behavior.

You can conceptualize this principle by likening it to a farmer passing down farming skills to his son, who should be capable of replacing his father if necessary. If the son decides to pursue farming, he can effectively take over his father's role. However, if he opts for a career in footbal, he cannot replace his father, despite both belonging to the same family hierarchy.

This principle extends the open-close principle and focuses on maintaining consistent behavior between a superclass and its subtypes.

The Liskov Substitution Principle



FACULTY OF COMPUTER SCIENCE

Source:https://medium.com/backticks-tildes/the-s-o-l-i-d-principles-in-pictures-b34ce2f1e898

The Liskov Substitution Principle

Rectangle.java

```
public class Rectangle {

        private double width;
        private double height;

        public double getWidth() {
                return width;
        }

        public void setWidth(double width) {
                this.width = width;
        }

        public double getHeight() {
                return height;
        }

        public void setHeight(double height) {
                this.height = height;
        }

        public double area(){
                return width*height;
        }
}
```

Violation

FACULTY OF COMPUTER SCIENCE

The Liskov Substitution Principle

Square.java

```
public class Square extends Rectangle{

    @Override
    public void setHeight(double height) {
        super.setHeight(height);
        super.setWidth(height);
    }

    @Override
    public void setWidth(double width) {
        super.setHeight(width);
        super.setWidth(width);
    }

}
```

Violation

FACULTY OF COMPUTER SCIENCE

The Liskov Substitution Principle

Rectangle.java

```
public class Rectangle {

    private double width;
    private double height;

    public Rectangle(double width, double height) {
        super();
        this.width = width;
        this.height = height;
    }

    public double getWidth() {
        return width;
    }

    public double getHeight() {
        return height;
    }

    public double area(){
        return width*height;
    }
}
```

Solution

FACULTY OF COMPUTER SCIENCE

The Liskov Substitution Principle

Square.java

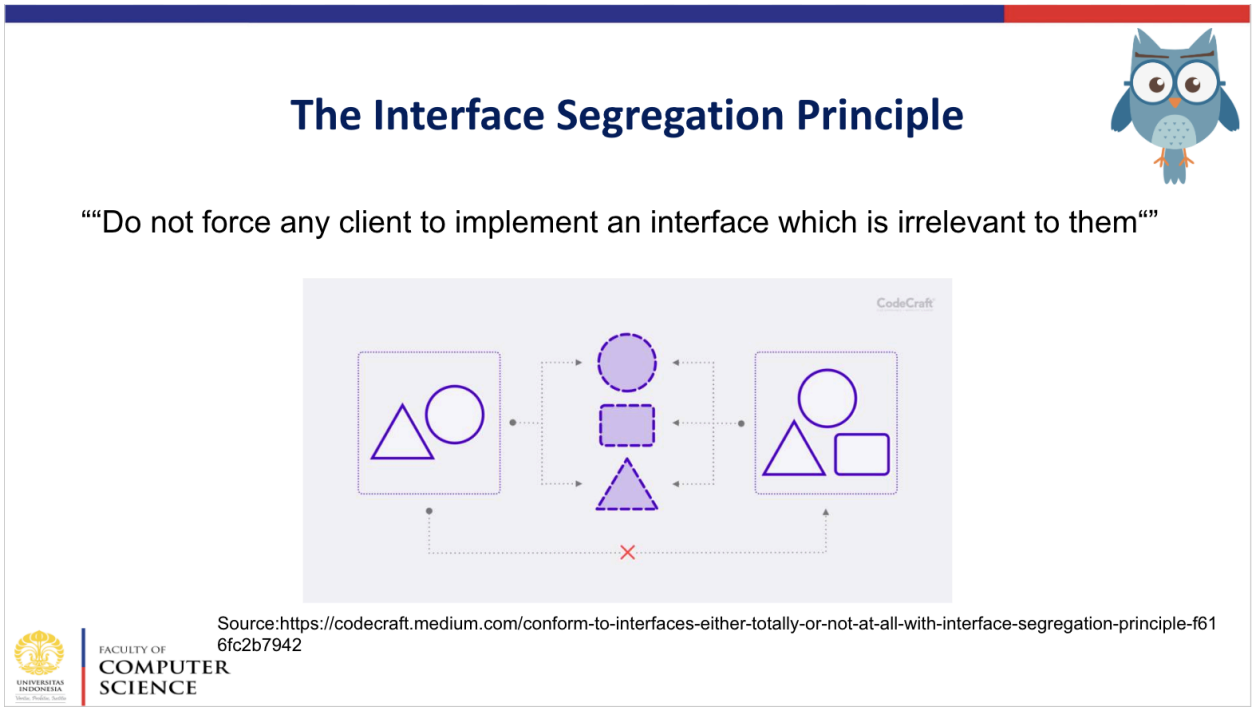
```
public class Square extends Rectangle{

    public Square(double side) {
          super(side, side);
     }
}
```

Solution

FACULTY OF COMPUTER SCIENCE

After understanding the explanation, please write in your own words, what is LSP?

Interface Segregation Principle (ISP)



The "I" in SOLID refers to interface segregation, emphasizing the subdivision of larger interfaces into smaller, more focused ones. This approach ensures that implementing classes are only burdened with methods pertinent to their functionality. Interface segregation, the first principle within SOLID that pertains specifically to interfaces rather than classes, shares similarities with the single responsibility principle. Essentially, it advises against compelling clients to implement irrelevant interfaces. It discourages imposing unnecessary methods on clients, promoting a more streamlined and efficient interface design.

The objective is avoidance of overly complex interfaces, encouraging the creation of numerous smaller interfaces tailored to specific clients. Rather than having one comprehensive interface, it's advocated to have multiple interfaces, each catering to distinct client needs with specific responsibilities.

The Interface Segregation Principle

Switches.java

```
public interface Switches {
   void startEngine();
   void shutDownEngine();
   void turnRadioOn();
   void turnRadioOff();
   void turnCameraOn();
   void turnCameraOff();
}
```

Violation

FACULTY OF COMPUTER SCIENCE

The Interface Segregation Principle

Vehicle.java

```
public abstract class Vehicle implements Switches {

   private boolean engineRunning;

   public boolean isEngineRunning() {
      return engineRunning;
   }

   @Override
   public void startEngine() {
      engineRunning = true;
   }

   @Override
   public void shutDownEngine() {
      engineRunning = false;
   }

}
```

Violation

FACULTY OF COMPUTER SCIENCE

The Interface Segregation Principle

Car.java

```
public class Car extends Vehicle {

    private boolean radioOn;

    public boolean isRadioOn() {
        return radioOn;
    }

    @Override
    public void turnRadioOn() {
        radioOn = true;
    }

    @Override
    public void turnRadioOff() {
        radioOn = false;
    }

    @Override
    public void turnCameraOn() {
        // nothing to do here
    }

    @Override
    public void turnCameraOff() {
        // nothing to do here
    }
}
```

Violation

FACULTY OF COMPUTER SCIENCE

The Interface Segregation Principle

Drone.java

```
public class Drone extends Vehicle {

    private boolean cameraOn;

    public boolean isCameraOn() {
        return cameraOn;
    }

    @Override
    public void turnCameraOn() {
        cameraOn = true;
    }

    @Override
    public void turnCameraOff() {
        cameraOn = false;
    }

    @Override
    public void turnRadioOn() {
        // nothing to do here
    }

    @Override
    public void turnRadioOff() {
        // nothing to do here
    }
}
```

Violation

FACULTY OF COMPUTER SCIENCE

The Interface Segregation Principle

CameraSwitch.java

```
public interface CameraSwitch {
    void turnCameraOn();
    void turnCameraOff();
}
```

Solution

FACULTY OF COMPUTER SCIENCE

The Interface Segregation Principle

EngineSwitch.java

```
public interface EngineSwitch {
    void startEngine();
    void shutDownEngine();
}
```

Solution

FACULTY OF COMPUTER SCIENCE

The Interface Segregation Principle

RadioSwitch.java

```
public interface RadioSwitch {

    void turnRadioOn();

    void turnRadioOff();
}
```

Solution

FACULTY OF COMPUTER SCIENCE

The Interface Segregation Principle

Vehicle.java

```
public abstract class Vehicle implements EngineSwitch {

    private boolean engineRunning;

    public boolean isEngineRunning() {
        return engineRunning;
    }

    @Override
    public void startEngine() {
        engineRunning = true;
    }

    @Override
    public void shutDownEngine() {
        engineRunning = false;
    }

}
```

Solution

FACULTY OF COMPUTER SCIENCE

The Interface Segregation Principle

Car.java

```
public class Car extends Vehicle implements RadioSwitch {

    private boolean radioOn;

    public boolean isRadioOn() {
        return radioOn;
    }

    @Override
    public void turnRadioOn() {
        radioOn = true;
    }

    @Override
    public void turnRadioOff() {
        radioOn = false;
    }

}
```

Solution

FACULTY OF COMPUTER SCIENCE

The Interface Segregation Principle

Drone.java

```
public class Drone extends Vehicle implements CameraSwitch {

    private boolean cameraOn;

    public boolean isCameraOn() {
        return cameraOn;
    }

    @Override
    public void turnCameraOn() {
        cameraOn = true;
    }

    @Override
    public void turnCameraOff() {
        cameraOn = false;
    }
}
```

Solution

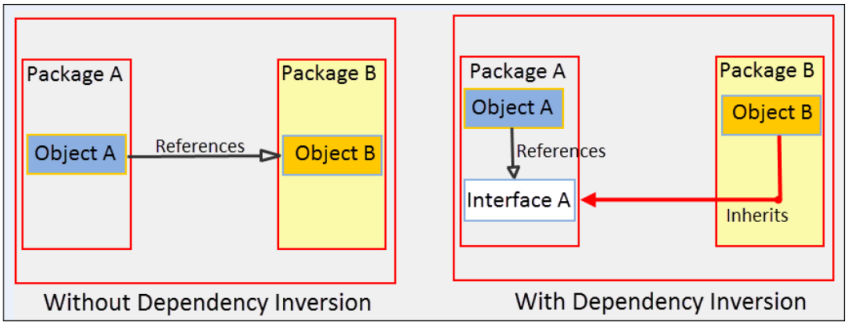
FACULTY OF COMPUTER SCIENCE

After understanding the explanation, please write in your own words, what is ISP?

Dependency Inversions Principle (DIP)

The Dependency Inversion Principle

"Our classes should depend upon interfaces or abstract classes instead of concrete classes and functions."



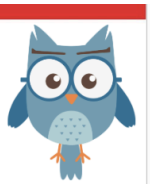
Source:https://springframework.guru/principles-of-object-oriented-design/dependency-inversion-principle/

FACULTY OF COMPUTER SCIENCE

The principle emphasizes the use of abstraction, such as abstract classes and interfaces, over concrete implementations. It suggests that high-level modules should not rely on low-level modules directly; instead, both should depend on abstraction. Abstraction, being independent of specific details, facilitates decoupling within the software, as opposed to the dependence of details on abstraction. This approach effectively separates concerns and enhances flexibility. Developers should depend on abstractions, not on concretions. It also instructs that high-level

module should not depend upon low-level modules. And, abstractions should not depend on details, but Details should depend upon abstractions.

The Dependency Inversion Principle



RacingCar.java

```
public class RacingCar {

    private final int maxFuel;
    private int remainingFuel;
    private int power;

    public RacingCar(final int maxFuel) {
        this.maxFuel = maxFuel;
        remainingFuel = maxFuel;
    }

    public void accelerate(){
        power++;
        remainingFuel--;
    }

}
```

Violation

FACULTY OF COMPUTER SCIENCE

The Dependency Inversion Principle

Pilot.java

```
public class Pilot {

    private RacingCar vehicle;

    public Pilot(){
        this.vehicle = new RacingCar(100);
    }

    public void increaseSpeed(){
        vehicle.accelerate();
    }
}
```

Violation

FACULTY OF COMPUTER SCIENCE

The Dependency Inversion Principle

RacingCar.java

```
public class RacingCar implements Vehicle{

    private final int maxFuel;
    private int remainingFuel;
    private int power;

    public RacingCar(final int maxFuel) {
        this.maxFuel = maxFuel;
        remainingFuel = maxFuel;
    }

    @Override
    public void accelerate() {
        power++;
        remainingFuel--;
    }
}
```

Solution

The Dependency Inversion Principle

Driver.java

```
public class Driver {

    private Vehicle vehicle;

    public Driver(final Vehicle vehicle){
        this.vehicle = vehicle;
    }

    public void increaseSpeed(){
        vehicle.accelerate();
    }
}
```

Solution

The Dependency Inversion Principle

Vehicle.java

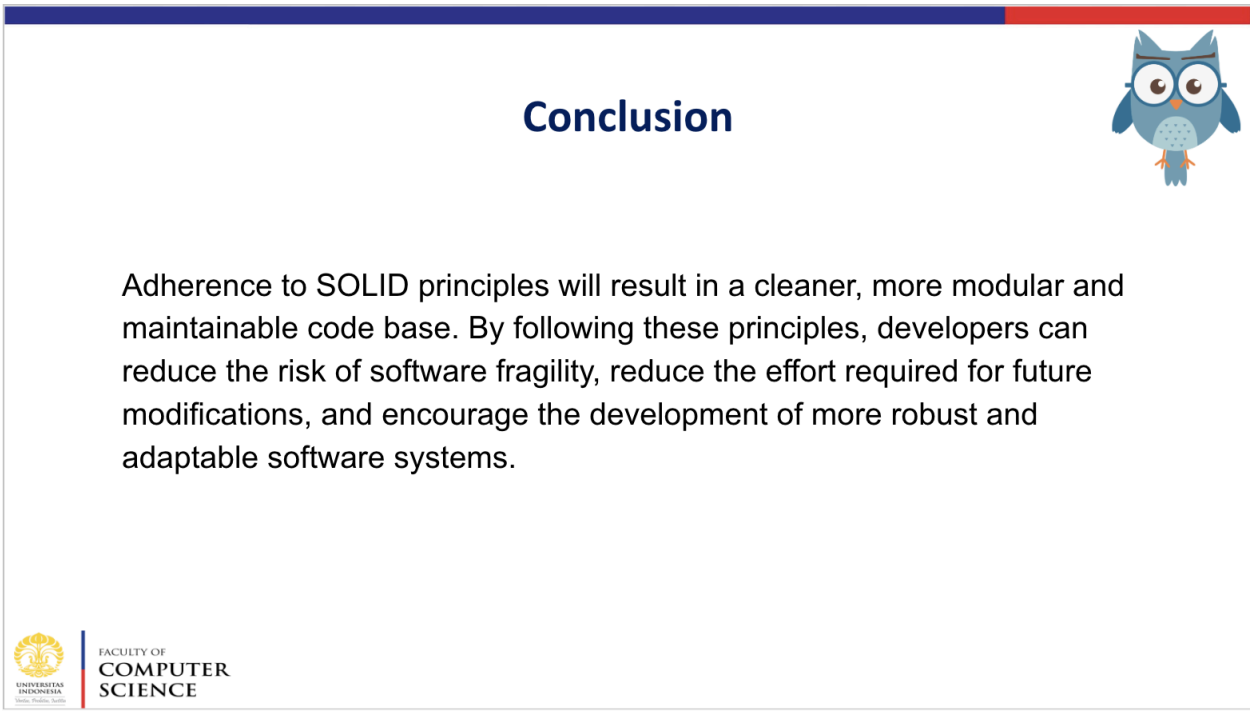
```
public interface Vehicle {
    void accelerate();
}
```

Solution

FACULTY OF COMPUTER SCIENCE

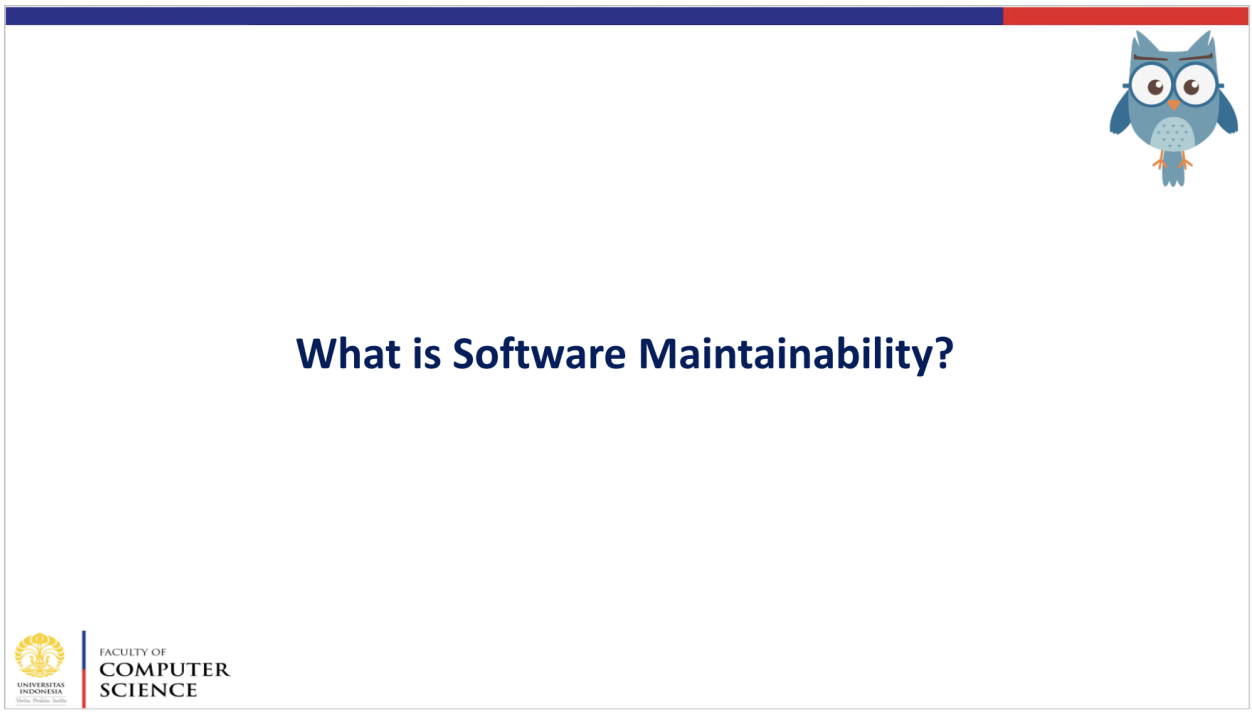
After understanding the explanation, please write in your own words, what is DIP?

Conclusion about SOLID Principles



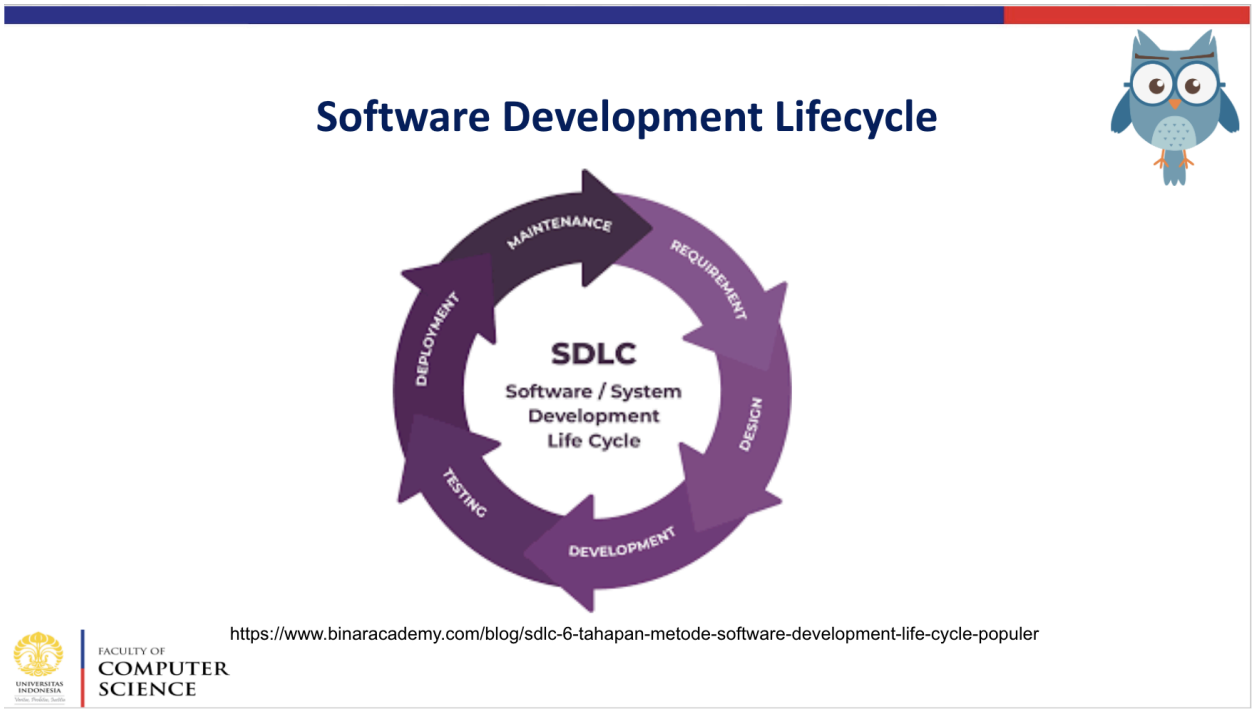
Adherence to SOLID principles will result in a cleaner, more modular and maintainable code base. By following these principles, developers can reduce the risk of software fragility, reduce the effort required for future modifications, and encourage the development of more robust and adaptable software systems. Therefore, it is important for developers to understand how to apply the principles to the software that will be developed later.

2. Software Maintainability



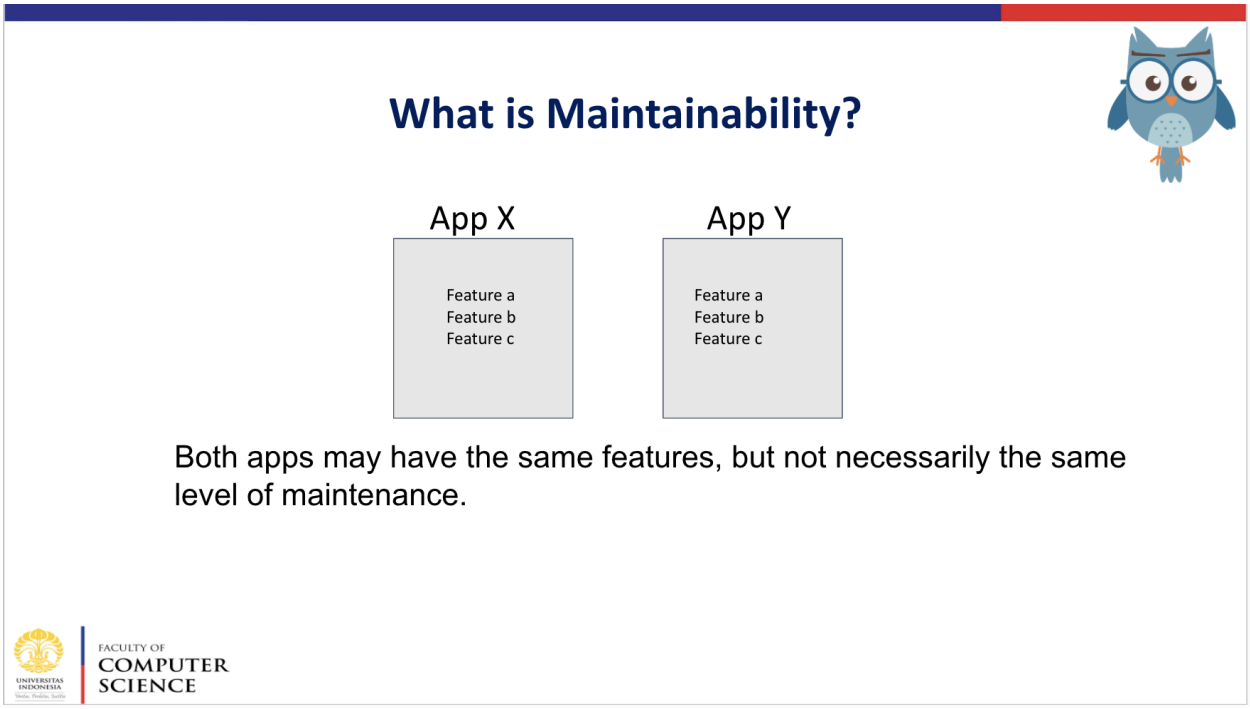
Please explain what you know about software maintainability!

Software Development Life Cycle



Throughout the SDLC, considerations for maintainability should be integrated into each phase. This includes prioritizing clarity, simplicity, modularity, and documentation, as well as implementing strategies for code reuse, version control, and continuous integration. By addressing maintainability throughout the SDLC, organizations can minimize technical debt, reduce future maintenance costs, and enhance the longevity of their software products. Maintenance plays a critical role in the Software Development Life Cycle (SDLC), constituting a significant phase following deployment. It encompasses various activities aimed at ensuring the continued functionality, performance, and relevance of the software over its operational lifespan.

What is Maintainability?



Imagine two different software systems that have exactly the same functionality. Given the same input, both compute exactly the same output. One of these two systems is fast and user-friendly, and its source code is easy to modify. The other system is slow and difficult to use, and its source code is nearly impossible to understand, let alone modify. Even though both systems have the same functionality, their quality clearly differs. Maintainability (how easily a system can be modified) is one characteristic of software quality. Performance (how slow or fast a system produces its output) is another.

What is Maintainability?

More formally, the IEEE Standard Glossary of Software Engineering Terminology defines maintainability as:

"The ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment."

FACULTY OF COMPUTER SCIENCE

The maintainability of software depends on a few different factors. In general, it must be easy to understand the software (how it works, what it does, and why it does it the way it does), easy to find what needs to be change, easy to make changes and easy to check that the changes have not introduced any bugs.

Characteristic of Maintainability Software

There are many factors to consider when building maintainable software. These include:

- **Readability**: The code should be easy to read and understand. This makes it easier for new developers to jump in and start working on the project, and also makes it easier to spot potential bugs.
- **Modularity**: The code should organize into logical modules that can be independently maintained and updated. This allows changes to be made without affecting the rest of the codebase and makes it easier to reuse code in other projects.
- **Testability**: The code should be written in a way that makes it easy to write automated tests. This helps ensure that new changes don't break existing functionality, and also makes it easier to catch bugs before they make it into production.
- **Flexibility**: The code should be flexible enough to accommodate changing requirements over time. This includes using abstractions and design patterns that make the code more resilient to change and avoiding tightly-coupled dependencies that make it difficult to modify individual components.
- **Scalability**: The code should design for scalability from the outset. This means considering things like performance, caching, load balancing, and other factors that can impact the system's ability to handle increased traffic or data volume.

FACULTY OF COMPUTER SCIENCE

Maintainability is a key characteristic of software quality, focusing on how easy it is to maintain, modify, and extend a software system. Here are some essential characteristics of maintainable software:

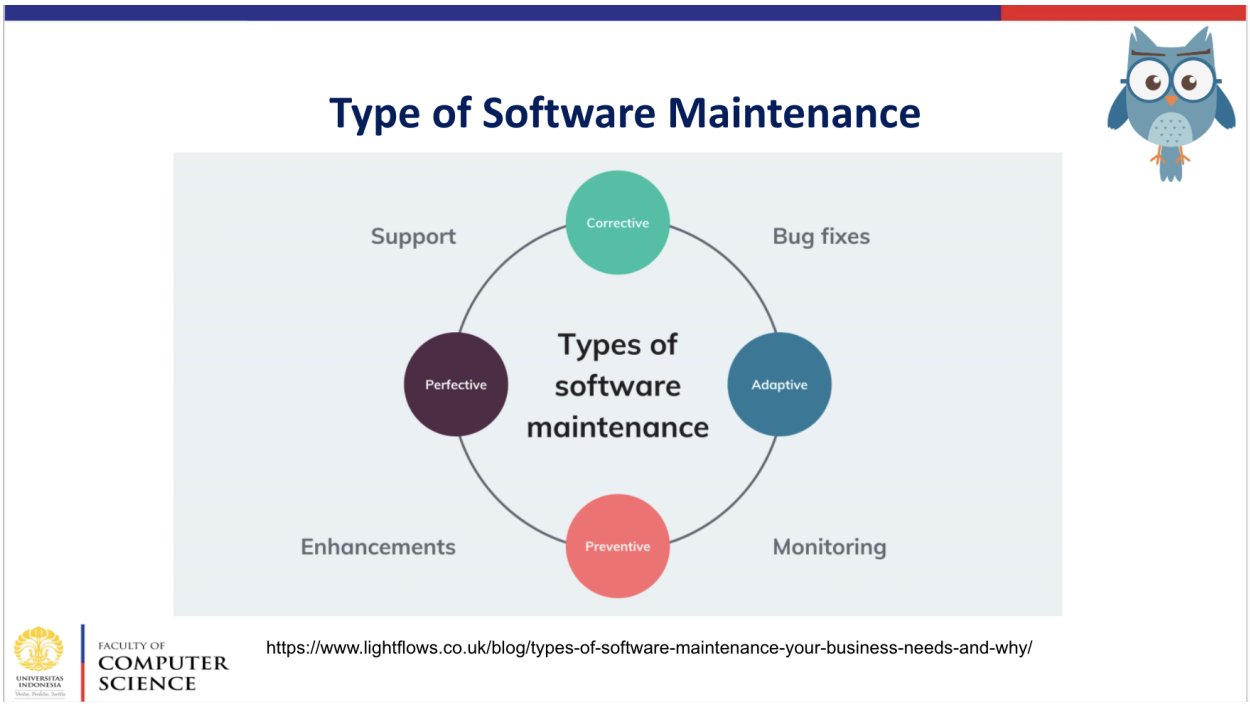
Readability: The code should be easy to read and understand. This makes it easier for new developers to jump in and start working on the project, and also makes it easier to spot potential bugs.

Modularity: The code should organize into logical modules that can be independently maintained and updated. This allows changes to be made without affecting the rest of the codebase and makes it easier to reuse code in other projects.

Testability: The code should be written in a way that makes it easy to write automated tests. This helps ensure that new changes don't break existing functionality, and also makes it easier to catch bugs before they make it into production.

Flexibility: The code should be flexible enough to accommodate changing requirements over time. This includes using abstractions and design patterns that make the code more resilient to change and avoiding tightly-coupled dependencies that make it difficult to modify individual components.

Scalability: The code should design for scalability from the outset. This means considering things like performance, caching, load balancing, and other factors that can impact the system's ability to handle increased traffic or data volume.



There are four types of software maintenance: Bugs are discovered and have to be fixed (this is called corrective maintenance). The system has to be adapted to changes in the environment in

which it operates—for example, upgrades of the operating system or technologies (this is called adaptive maintenance). Users of the system (and/or other stakeholders) have new or changed requirements (this is called perfective maintenance). Ways are identified to increase quality or prevent future bugs from occurring (this is called preventive maintenance).

Why is Maintainability Important?

Please explain why maintenance is important!

Why Is Maintainability Important?

There are two angles to this question:

- Maintainability has significant business impact.
- Maintainability is an enabler for other quality characteristics.

FACULTY OF COMPUTER SCIENCE

As you've discovered, while maintainability constitutes just one aspect of the eight quality characteristics outlined in ISO 25010, its significance prompts the need for an entire book dedicated to its exploration. This raises a dual perspective on the matter: Maintainability has significant business impact and maintainability is an enabler for other quality characteristics.

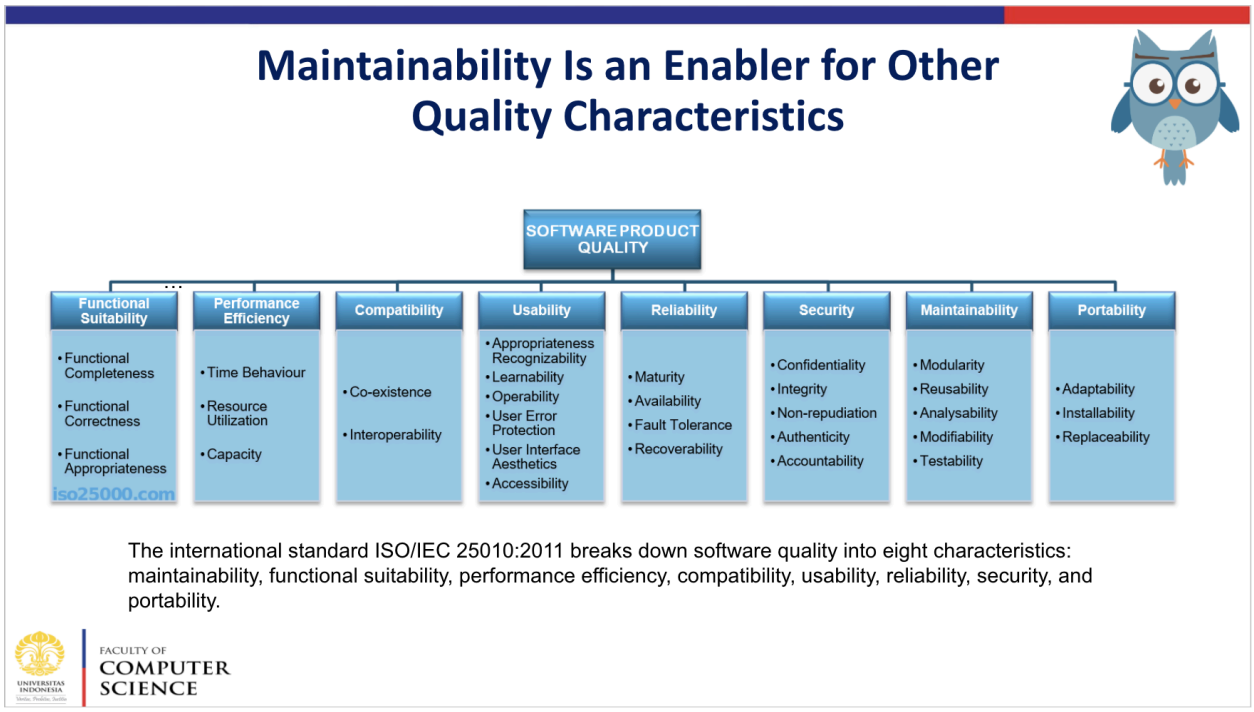
Maintainability Has Significant Business Impact

FACULTY OF COMPUTER SCIENCE

In software development, the maintenance phase of a software system often extends over a decade or more. Throughout this prolonged period, a continual flow of issues requiring resolution (corrective and adaptive maintenance) and enhancement requests needing fulfillment (perfective maintenance) persists. The ability to swiftly and effectively address these issues and implement enhancements holds significant importance for stakeholders.

Efficient and effective maintenance practices result in reduced efforts when it comes to issue resolution and enhancements. This reduction translates to lower maintenance costs, particularly if it leads to a decrease in the number of personnel (developers) required for maintenance tasks. Moreover, when maintenance tasks can be completed efficiently, developers have more time available for other responsibilities, such as developing new features.

The expeditious implementation of enhancements also carries benefits, such as shorter time-to-market for new products and services supported by the system. Conversely, if issue resolution and enhancements are slow or cumbersome, there is a risk of missing deadlines or rendering the system unusable.

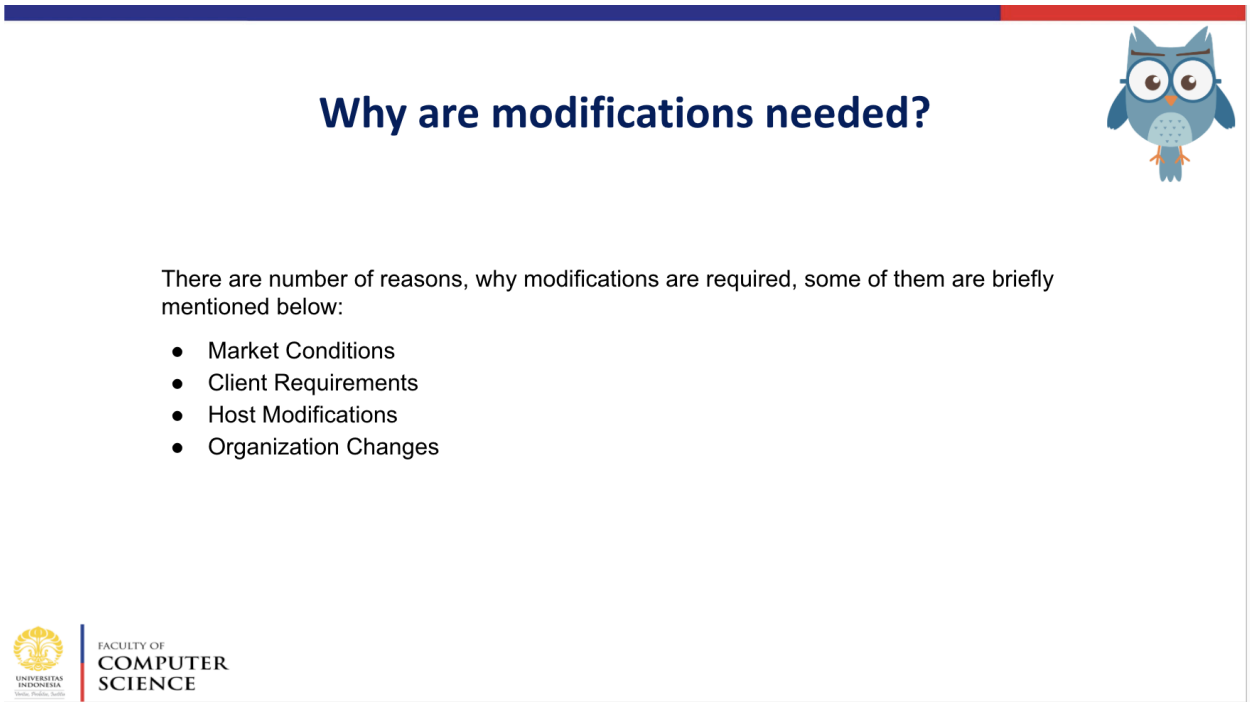


Another rationale behind the unique significance of maintainability within software quality lies in its role as a facilitator for other quality attributes. When a system exhibits high maintainability, the process of enhancing other quality aspects, such as rectifying a security vulnerability, becomes more straightforward. In broader terms, enhancing a software system typically entails

modifications to its source code, whether for improving performance, functional adequacy, security, or any other of the seven quality characteristics outlined by ISO 25010.

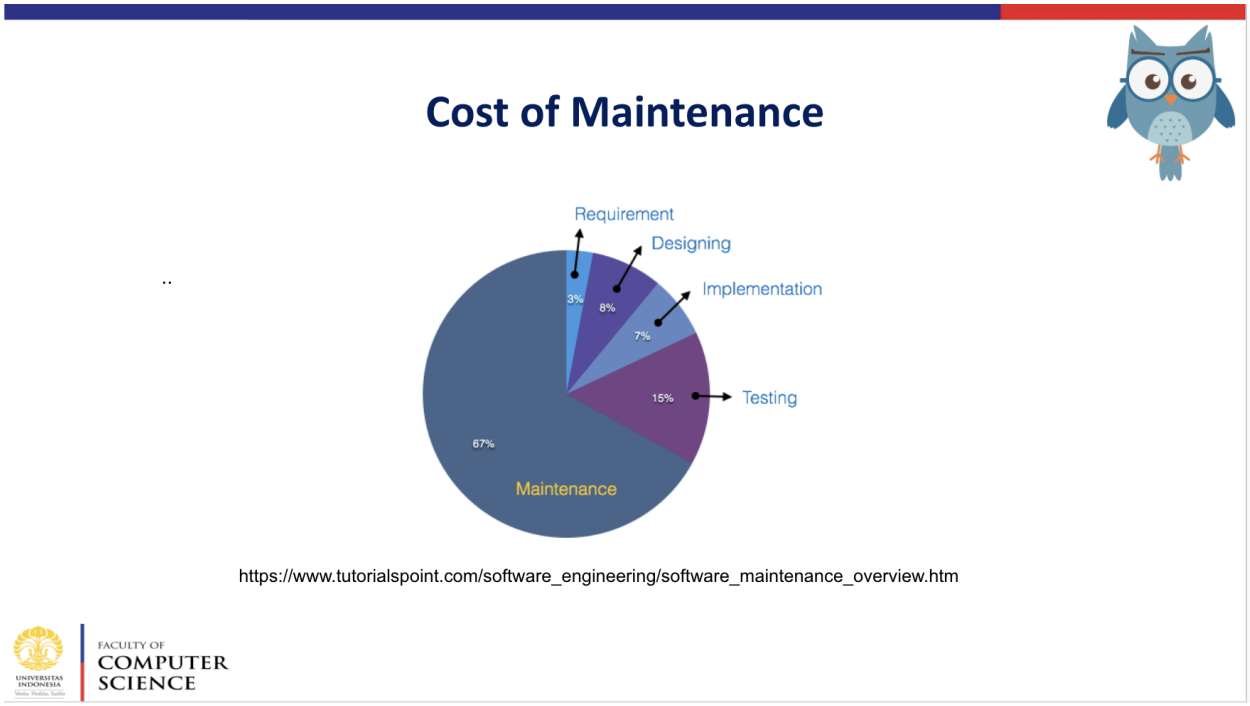
These modifications range from minor, localized changes to more extensive restructuring efforts. Regardless of their scale, all modifications necessitate locating specific sections of source code, comprehending their underlying logic, understanding their integration within the system's operational workflow, analyzing dependencies between various code components, testing them, and integrating them into the development pipeline. In systems characterized by higher maintainability, these modifications are comparatively easier to execute, facilitating the implementation of quality enhancements with greater speed and efficacy.

For instance, code with high maintainability exhibits greater stability than its unmaintainable counterparts: alterations made within a highly maintainable system tend to produce fewer unexpected side effects compared to changes made within a convoluted system that is challenging to analyze and test.



Market conditions, including evolving policies such as taxation regulations and newly imposed constraints like updated bookkeeping standards, can necessitate modifications to software. Client requirements may evolve over time, leading to requests for new features or functionalities in the software. Changes in the hardware or platform of the target host, such as updates to the operating system, may require modifications to ensure compatibility. Organizational changes at

the client's end, such as workforce reductions, acquisitions, or expansion into new business areas, may prompt the need for modifications to the original software.



Reports show that maintenance costs are high. A study on software maintenance estimation found that maintenance costs account for 67% of the cost of the entire software process cycle. Therefore, we need to pay more attention to maintenance

Real Word Affecting Cost of Maintenance

- The standard age of any software is considered up to 10 to 15 years.
- Older softwares, which were meant to work on slow machines with less memory and storage capacity cannot keep themselves challenging against newly coming enhanced softwares on modern hardware.
- As technology advances, it becomes costly to maintain old software.
- Most maintenance engineers are newbie and use trial and error method to rectify problem.
- Often, changes made can easily hurt the original structure of the software, making it hard for any subsequent changes.
- Changes are often left undocumented which may cause more conflicts in future.

UNIVERSITAS INDONESIA
FACULTY OF COMPUTER SCIENCE

The standard age of any software is considered up to 10 to 15 years.

Older softwares, which were meant to work on slow machines with less memory and storage capacity cannot keep themselves challenging against newly coming enhanced softwares on modern hardware.

As technology advances, it becomes costly to maintain old software.

Most maintenance engineers are newbie and use trial and error method to rectify problem.

Often, changes made can easily hurt the original structure of the software, making it hard for any subsequent changes.

Changes are often left undocumented which may cause more conflicts in future.

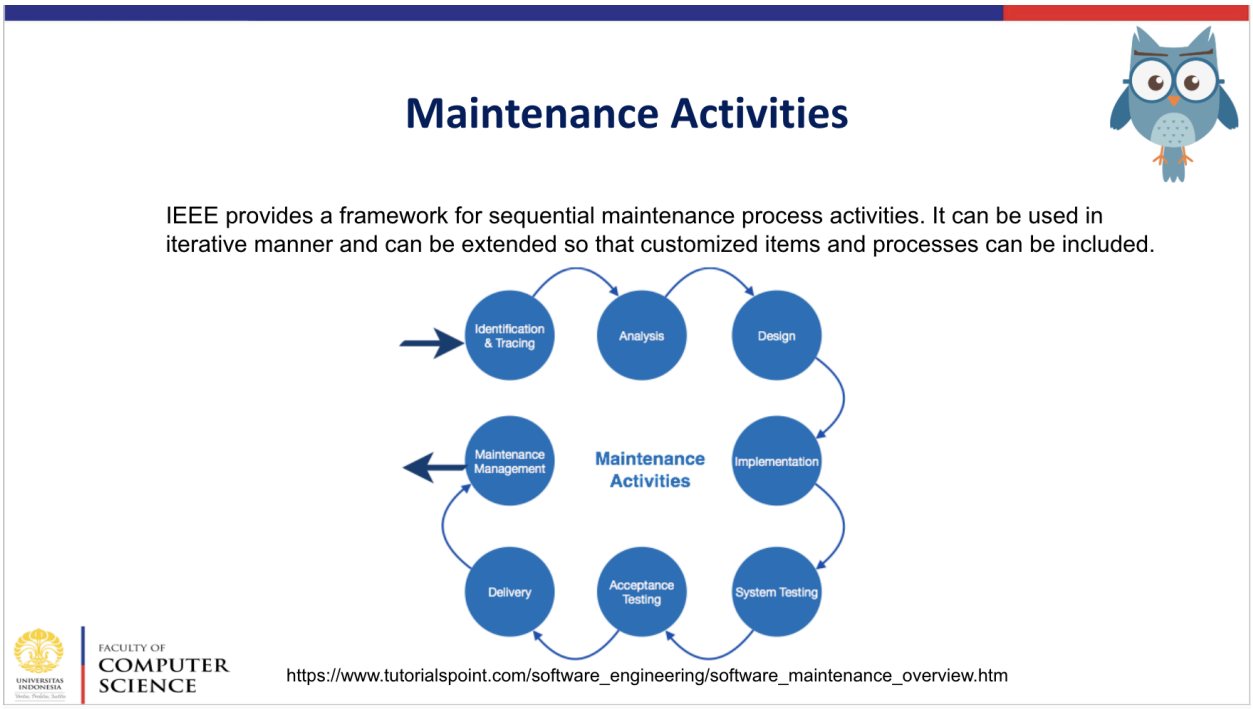
Software-end factors affecting Maintenance Cost

- Structure of Software Program
- Programming Language
- Dependence on external environment
- Staff reliability and availability

FACULTY OF COMPUTER SCIENCE

Software factors that affect Maintenance Costs include:

- Structure of Software Program
- Programming Language
- Dependence on external environment
- Staff reliability and availability



Identification & Tracing - This phase involves activities aimed at identifying the need for modifications or maintenance. Users may generate requests for changes, or the system itself may report issues through logs or error messages. Additionally, this phase involves classifying the type of maintenance required.

Analysis - Modifications are carefully analyzed to assess their impact on the system, including considerations of safety and security implications. If the potential impact is significant, alternative solutions are explored. The identified modifications are then translated into detailed requirement specifications, and an analysis of the associated modification/maintenance costs is conducted, concluding with estimation.

Design - New modules that require replacement or modification are designed based on the requirement specifications established in the previous phase. Test cases are developed to validate and verify the functionality of the new modules.

Implementation - The new modules are coded based on the structured design created in the design phase. Each programmer is responsible for conducting unit testing concurrently with coding.

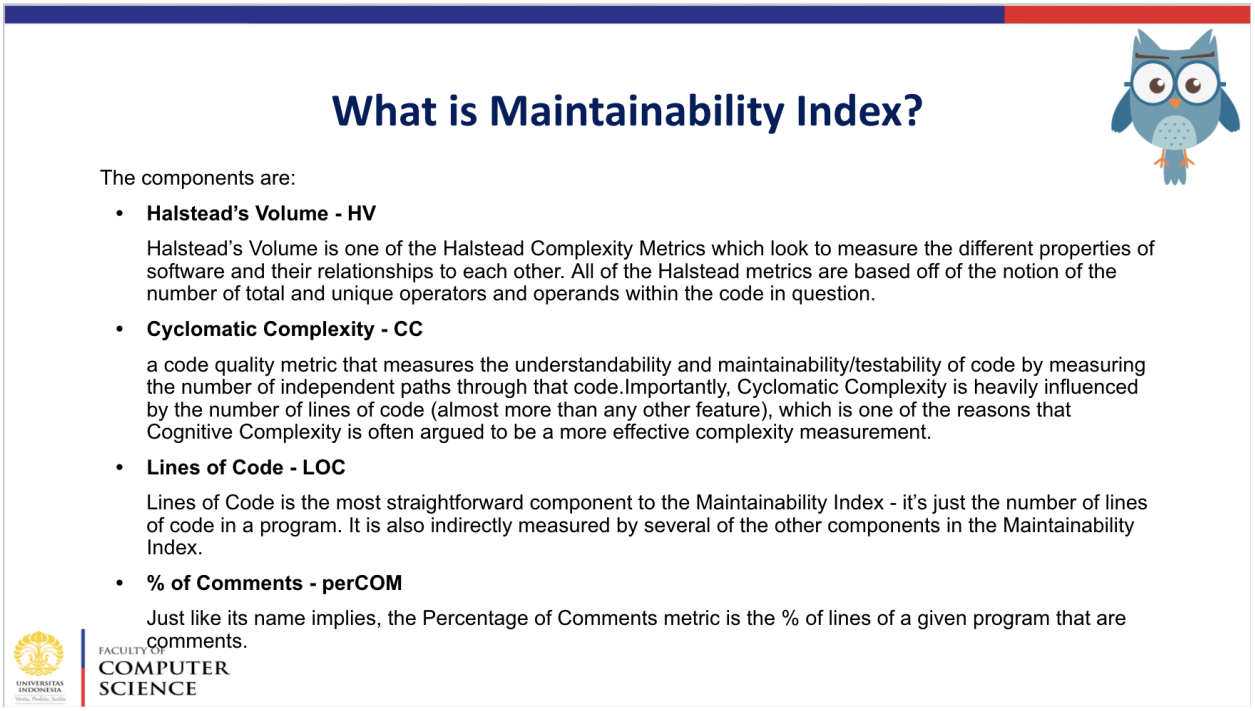
System Testing - Integration testing is performed to ensure seamless interaction among the newly created modules. Additionally, integration testing is conducted between the new modules and the existing system. Finally, the system undergoes comprehensive testing as a whole, including regression testing procedures.

Acceptance Testing - Following internal testing, the system is subjected to acceptance testing with the participation of users. Any issues identified during this phase are addressed or noted for future iterations.

Delivery - Upon successful acceptance testing, the system is deployed throughout the organization, either through incremental updates or a fresh installation. Final testing occurs at the client's end after the software is delivered. If necessary, training facilities are provided, along with the distribution of hard copies of the user manual.

Maintenance Management - Configuration management plays a crucial role in system maintenance. Version control tools are utilized to manage different versions of the software, as well as for semi-version or patch management purposes.

How to Measure Software Maintainability?



The concept of the Maintainability Index was introduced in 1992 by Paul Oman and Jack Hagemeister. Their objective was to introduce automated software development metrics to assist in making decisions related to software development.

What Does the Maintainability Index Mean?

There are general score range guidelines:

=> 85 - Highly Maintainable

65 - 85 - Moderately Maintainable

<= 65 - Difficult to Maintain

Under the new Visual Studio definition the ranges changed slightly:

=> 20 - Highly Maintainable

=> 10 & < 20 - Moderately Maintainable

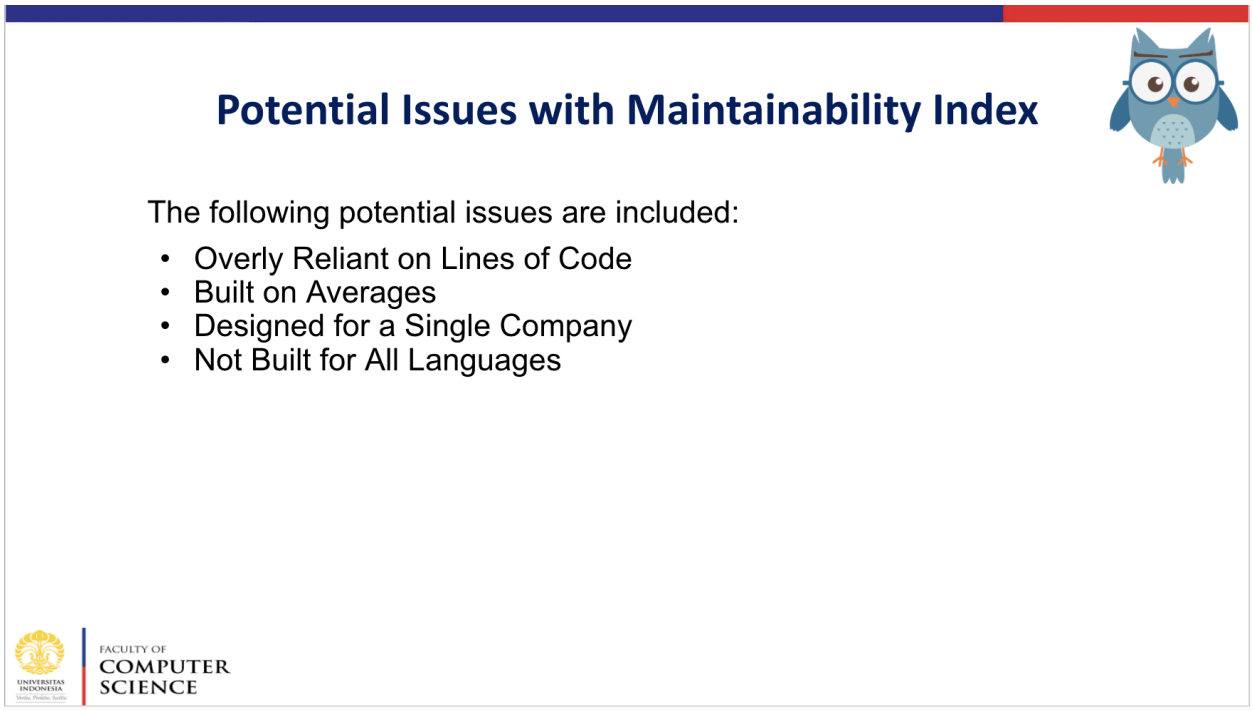
<10 - Difficult to Maintain

Both of these formulas and threshold ranges are used by a variety of tools

UNIVERSITAS INDONESIA

FACULTY OF COMPUTER SCIENCE

The original Maintainability Index had an upper bound of 171 and no lower bound. In general, in the original paper, the authors recommended that Maintainability Index primarily be used to calculate relative maintainability between sections of projects or projects overall for the same team - rather than be used as an absolute metric.



Overly Reliant on Lines of Code

The Maintainability Index heavily relies on the Lines of Code metric, which not only directly influences its calculation but also correlates strongly with Halstead Volume and Cyclomatic Complexity. Consequently, the index tends to overly emphasize the length of a file or the average length of files within a project. This can result in a decrease in the Maintainability Index even if changes to the code improve clarity and comprehensibility.

Built on Averages

Whether applied to an entire project or individual files, the Maintainability Index is computed based on the average values of Halstead Volume and Cyclomatic Complexity. However, research suggests that both complexity and maintainability typically follow a power law distribution. By using averages, the true impact of highly complex or costly functions, classes, and files in a codebase may be overlooked.

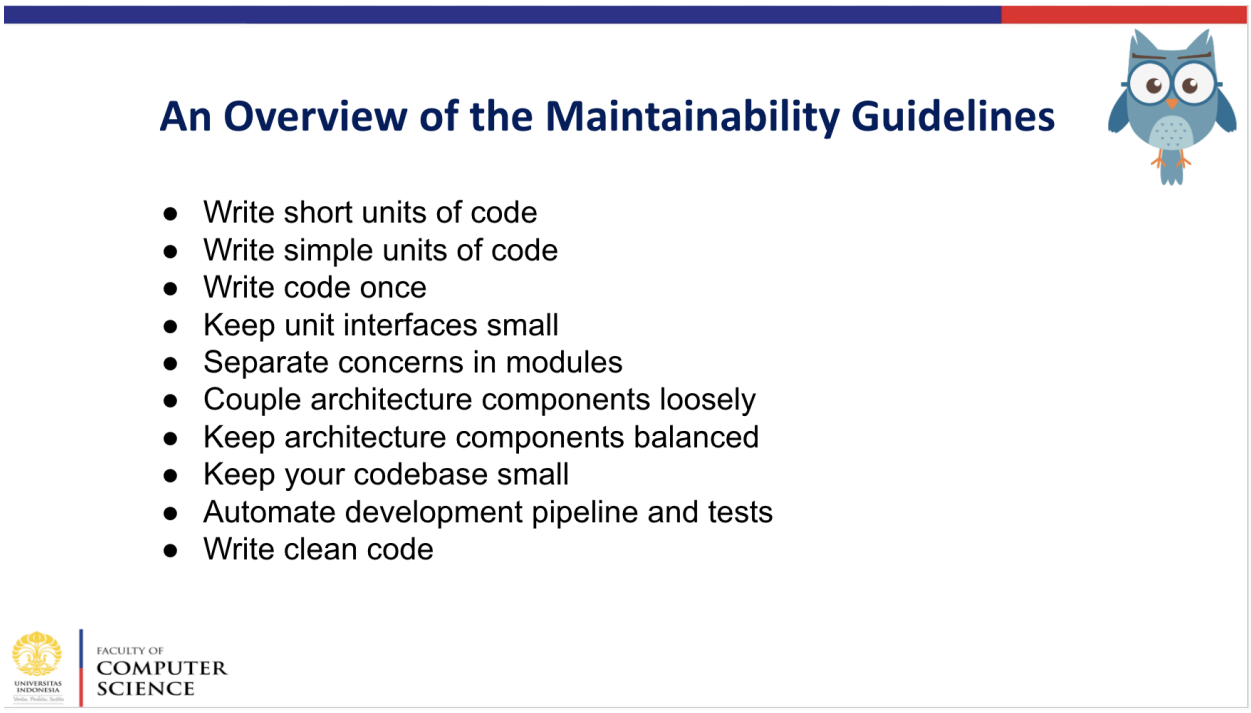
Designed for a Single Company

The formula for calculating the Maintainability Index was developed by engineers at HP for HP projects. While it may offer valuable insights for other projects, the coefficients and relationships within the formula have remained unchanged for over three decades, based on the initial estimates for maintainability.

Not Built for All Languages

The original formula for the Index was devised for projects written in C, yet it has been applied to other programming languages over time. However, there's uncertainty about whether the coefficients and formula are equally applicable across different languages and projects, as they were tailored specifically for C.

How to Achieve Maintainability?



Write short units of code

Shorter units (that is, methods and constructors) are easier to analyze, test, and reuse.

Write simple units of code

Units with fewer decision points are easier to analyze and test.

Write code once

Duplication of source code should be avoided at all times, since changes will need to be made in each copy. Duplication is also a source of regression bugs.

Keep unit interfaces small

Units (methods and constructors) with fewer parameters are easier to test and reuse.

Separate concerns in modules

Modules (classes) that are loosely coupled are easier to modify and lead to a more modular system.

Couple architecture components loosely

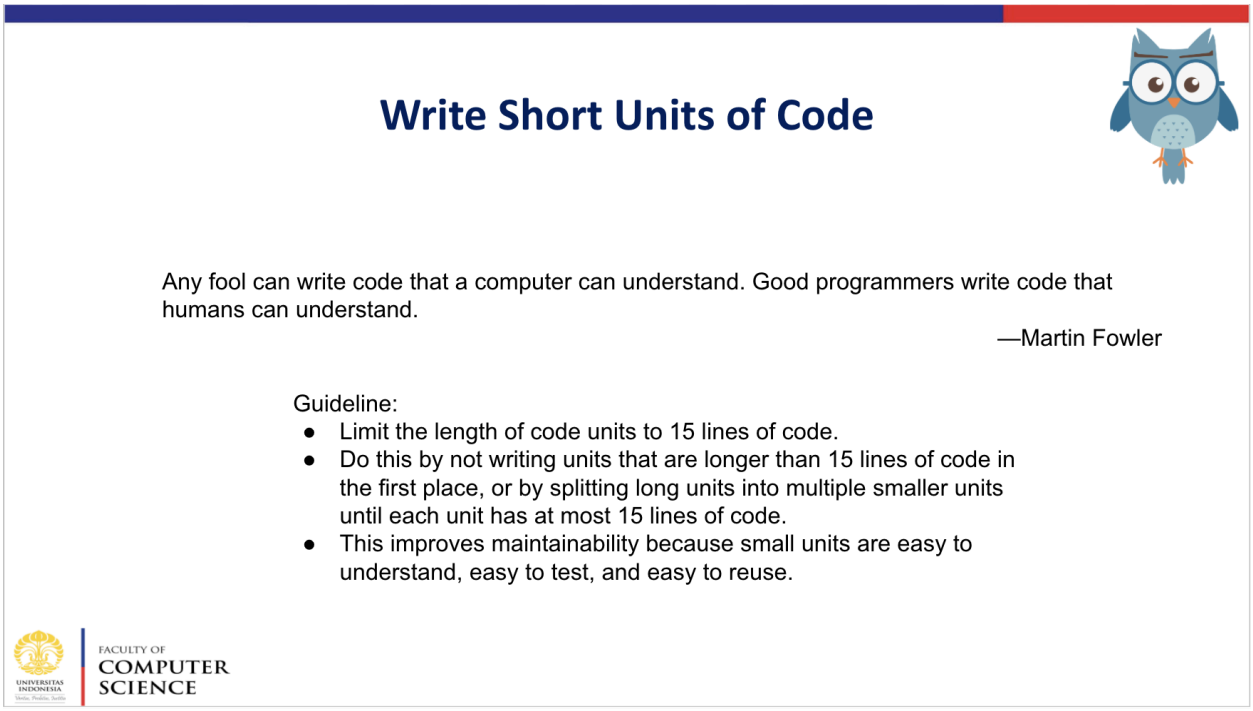
Top-level components of a system that are more loosely coupled are easier to modify and lead to a more modular system.

Keep architecture components balanced

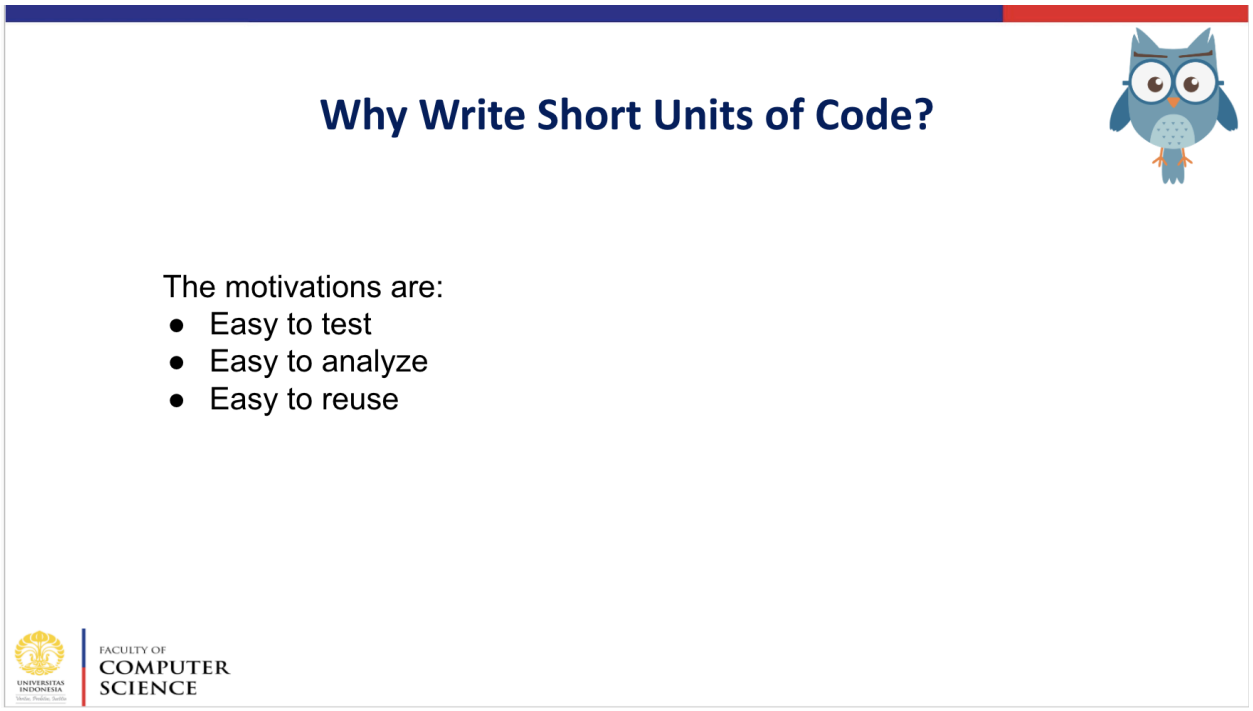
A well-balanced architecture, with not too many and not too few components, of uniform size, is the most modular and enables easy modification through separation of concerns.

Keep your codebase small

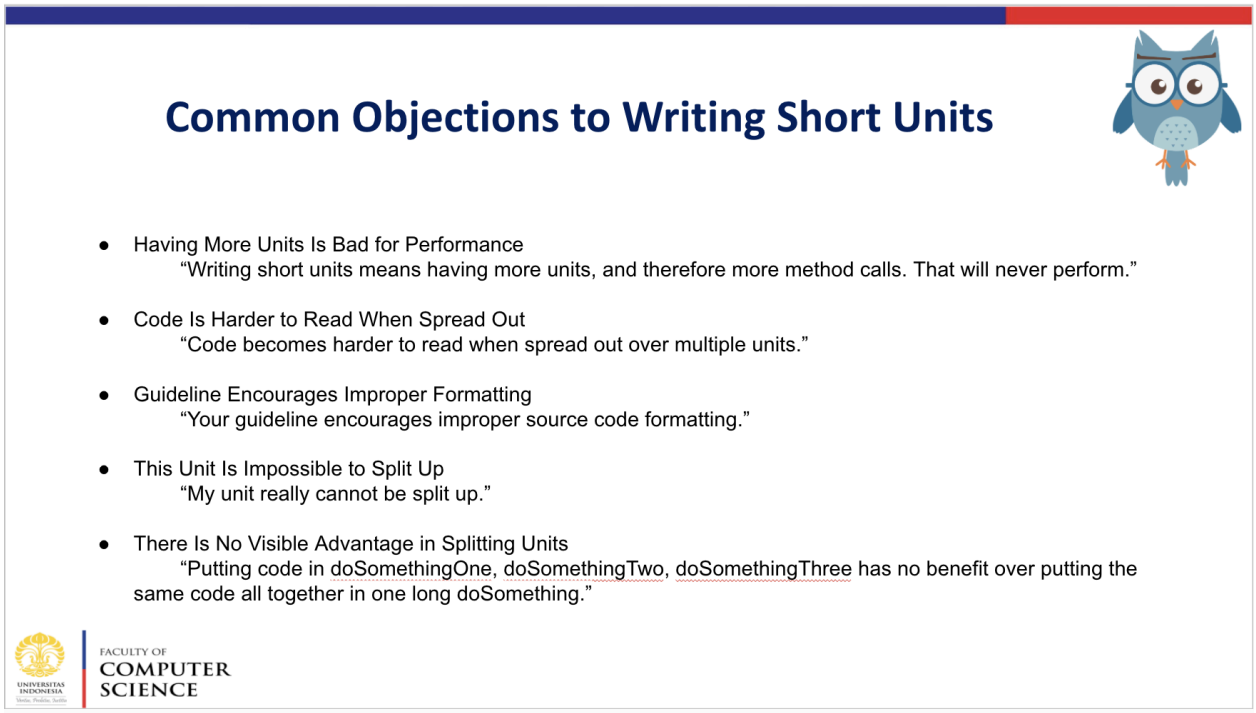
A large system is difficult to maintain, because more code needs to be analyzed, changed, and tested. Also, maintenance productivity *per line of code* is lower in a large system than in a small system.



Units refer to the smallest units of code that can be maintained and executed autonomously. These units typically consist of methods or constructors. When a unit is executed, it is executed in its entirety; it's not feasible to execute only a portion of a unit. Consequently, the smallest segment of code that can be reused and tested is a unit.



Easy to test. Units encapsulate the application logic, and significant testing effort is typically dedicated to verifying their correctness. Since the Java compiler and IDEs do not automatically detect logic errors, thorough testing is essential. Short units, with a single responsibility, are easier to test because they focus on a specific task. Easy to analyze. Reading and understanding the internal workings of a short unit requires less time compared to longer units. While this might not be immediately apparent during code creation, it significantly aids in modifying existing code during maintenance tasks, which often commence shortly after a project begins. Easy to reuse. Units should be invoked within methods to avoid being classified as dead code. Short units are more conducive to reuse as they tend to offer more generic functionality compared to longer units, which often have more specialized functionality. The generic nature of short units makes them more likely to fit various needs, promoting code reuse and keeping the overall codebase compact.



Having More Units Is Bad for Performance

"Writing short units means having more units, and therefore more method calls. That will never perform."

Indeed, theoretically, there is a performance penalty for having more units. There will be more method invocations (compared to having fewer, longer units). For each invo- cation, a bit of work needs to be done by the Java Virtual Machine (JVM). In practice, this is almost never a problem. In the worst case, we are talking about microseconds. Unless a unit is executed hundreds of thousands of times in a loop, the performance penalty of a method invocation is not noticeable. Also, the JVM is *very* good at opti- mizing the overhead of method invocations.

Code Is Harder to Read When Spread Out

"Code becomes harder to read when spread out over multiple units."

People have a working memory of about seven items, so someone who is reading a unit that is significantly longer than seven lines of code cannot process all of it. The exception is probably the original author of a piece of source code while he or she is working on it (but not a week later).

Guideline Encourages Improper Formatting

“Your guideline encourages improper source code formatting.”

Do not try to comply with guideline by cutting corners in the area of formatting. We are talking about putting multiple statements or multiple curly brackets on one line. It makes the code slightly harder to read and thus decreases its maintainability. Resist the temptation to do so.

This Unit Is Impossible to Split Up

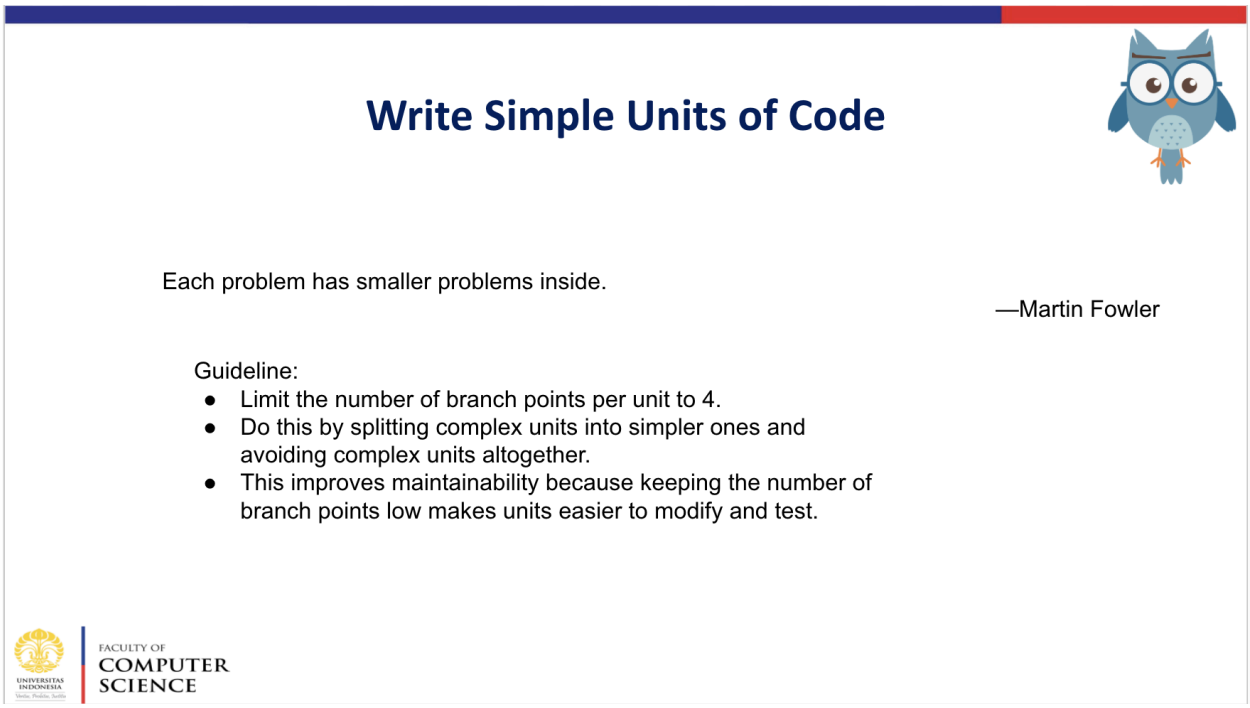
“My unit really cannot be split up.”

Sometimes, splitting a method is indeed difficult. Take, for instance, a properly for- matted switch statement in Java. For each case of the switch statement, there is a line for the case itself, at least one line to do anything useful, and a line for the break statement. So, anything beyond four cases becomes very hard to fit into 15 lines of code, and a case statement cannot be split

There Is No Visible Advantage in Splitting Units

“Putting code in doSomethingOne, doSomethingTwo, doSomethingThree has no benefit over putting the same code all together in one long doSomething.”

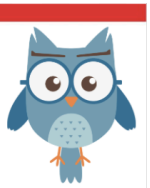
Actually, it does, provided you choose better names than doSomethingOne, doSome thingTwo, and so on. Each of the shorter units is, on its own, easier to understand than the long doSomething. More importantly, you may not even need to consider all the parts, especially since each of the method names, when chosen carefully, serves as documentation indicating what the unit of code is supposed to do. Moreover, the long doSomething typically will combine multiple tasks. That means that you can only reuse doSomething if you need the exact same combination. Most likely, you can reuse each of doSomethingOne, doSomethingTwo, and so on much more easily.



Complexity is an often disputed quality characteristic. Code that appears complex to an outsider or novice developer can appear straightforward to a developer that is completely familiar with it. To a certain extent, what is “complex” is in the eye of the holder. There is, however, a point where the code becomes so complex that modifying it becomes an extremely risky and very time-consuming task, let alone testing the modifications afterward. To keep code maintainable, we must put a limit on complexity. Another reason to measure complexity is knowing the minimum number of tests we need to be sufficiently certain that the system acts predictably. Before we can define such a code complexity limit, we must be able to measure complexity.

A common way to objectively assess complexity is to count the number of possible paths through a piece of code. The idea is that the more paths can be distinguished, the more complex a piece of code is. We can determine the number of paths unam- biguously by counting the number of branch points. A branch point is a statement where execution can take more than one direction depending on a condition. Exam- ples of branch points in Java code are if and switch statements.

Why Write Simple Units of Code?



Keeping your units simple is important for two main reasons:

- A simple unit is easier to understand, and thus modify, than a complex one.
- Simple units ease testing.

FACULTY OF COMPUTER SCIENCE

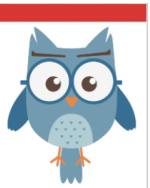
Simple Units Are Easier to Modify

Units with high complexity are generally hard to understand, which makes them hard to modify.

Simple Units Are Easier to Test

There is a good reason you should keep your units simple: to make the process of testing easier.

Common Objections to Writing Simple Units of Code



- High Complexity Cannot Be Avoided
  "Our domain is very complex, and therefore high code complexity is unavoidable."
- Splitting Up Methods Does Not Reduce Complexity
  "Replacing one method with McCabe 15 by three methods with McCabe 5 each means that overall McCabe is still 15 (and therefore, there are 15 control flow branches overall). So nothing is gained."

FACULTY OF COMPUTER SCIENCE

High Complexity Cannot Be Avoided

"Our domain is very complex, and therefore high code complexity is unavoidable."

Complexity in the domain does not require the technical implementation to be complex as well. In fact, it is your responsibility as a developer to simplify problems such that they lead to simple code. Even if the system as a whole performs complex functionality, it does not mean that units on the lowest level should be complex as well.

Splitting Up Methods Does Not Reduce Complexity

"Replacing one method with McCabe 15 by three methods with McCabe 5 each means that overall McCabe is still 15 (and therefore, there are 15 control flow branches overall).

So nothing is gained." Of course, you will not decrease the overall McCabe complexity of a system by refac- toring a method into several new methods. But from a maintainability perspective, there is an advantage to doing so: it will become easier to test and understand the code that was written. So, as we already mentioned, newly written unit tests allow you to more easily identify the root cause of your failing tests.

Write Code Once

Number one in the stink parade is duplicated code.
—Kent Beck and Martin Fowler, Bad Smells in Code

Guideline:
- Do not copy code.
- Do this by writing reusable, generic code and/or calling existing methods instead.
- This improves maintainability because when code is copied, bugs need to be fixed at multiple places, which is inefficient and error-prone.

FACULTY OF COMPUTER SCIENCE

Copying existing code looks like a quick win—why write something anew when it already exists? The point is: copied code leads to duplicates, and duplicates are a problem. As the quote above indicates, some even say that duplicates are the biggest software quality problem of all.

Why Write Code Once?

The main reasons are:
- Harder to Analyze
- Harder to Modify

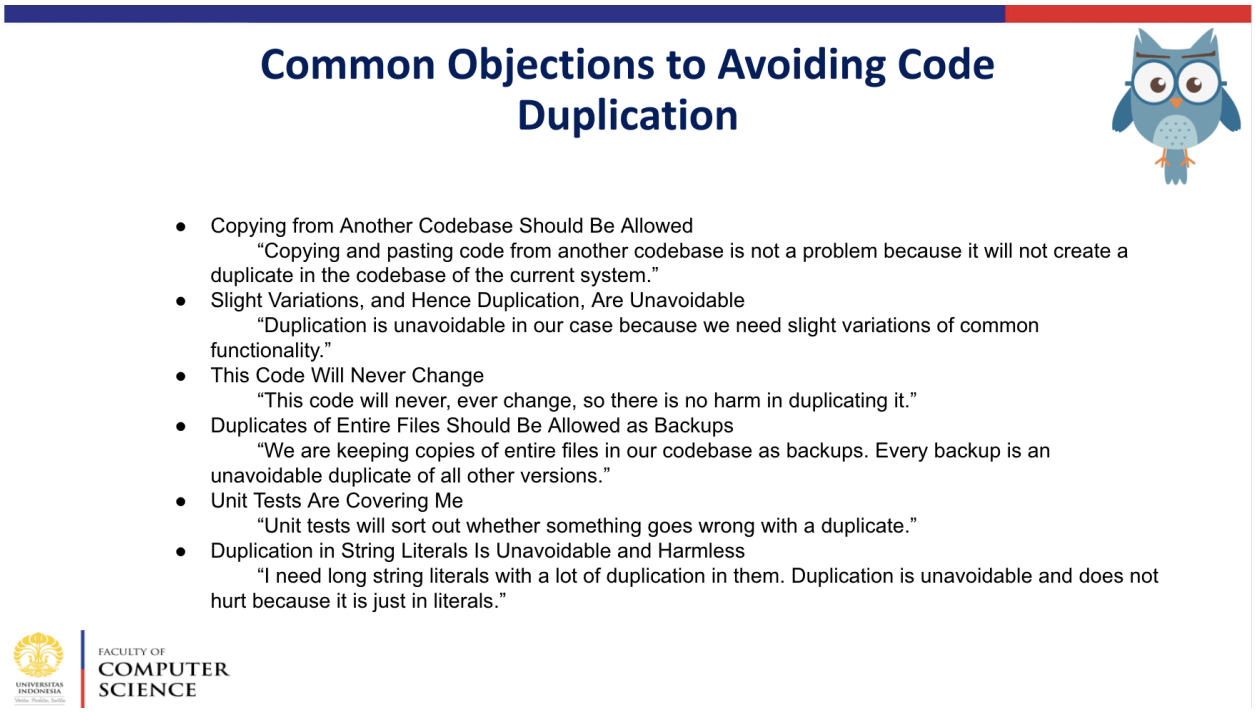
FACULTY OF COMPUTER SCIENCE

Duplicated Code Is Harder to Analyze

If you have a problem, you want to know how to fix it. And part of that “how” is where to locate the problem. When you are calling an existing method, you can easily find the source. When you are copying code, the source of the problem may exist elsewhere as well. However, the only way to find out is by using a clone detection tool.

Duplicated Code Is Harder to Modify

All code may contain bugs. But if duplicated code contains a bug, the same bug appears multiple times. Therefore, duplicated code is harder to modify; you may need to repeat bug fixes multiple times.



Copying from Another Codebase Should Be Allowed

“Copying and pasting code from another codebase is not a problem because it will not create a duplicate in the codebase of the current system.”

Technically, that is correct: it does not create a duplicate in the codebase of the current system. Copying code from another system may seem beneficial if the code solves the exact same problem in the exact same context.

Slight Variations, and Hence Duplication, Are Unavoidable

“Duplication is unavoidable in our case because we need slight variations of common functionality.”

Indeed, systems often contain slight variations of common functionality. For instance, some functionality is slightly different for different operating systems, for other versions (for reasons of backward compatibility), or for different customer groups. However, this does not imply that duplication is unavoidable.

This Code Will Never Change

“This code will never, ever change, so there is no harm in duplicating it.”

If it is absolutely, completely certain that code will never, ever change, duplication (and every other aspect of maintainability) is not an issue. For a start, you have to be absolutely, completely certain that the code in question also does not contain any bugs that need fixing.

Duplicates of Entire Files Should Be Allowed as Backups

“We are keeping copies of entire files in our codebase as backups. Every backup is an unavoidable duplicate of all other versions.”

We recommend keeping backups, but not in the way implied by this objection (inside the codebase). Version control systems such as SVN and Git provide a much better backup mechanism.

Unit Tests Are Covering Me

“Unit tests will sort out whether something goes wrong with a duplicate.”

This is true only if the duplicates are in the same method, and the unit test of the method covers both. If the duplicates are in other methods, it can be true only if a code analyzer alerts you if duplicates are changing. Otherwise, unit tests would not necessarily signal that something is wrong if only one duplicate has changed. Hence, you cannot rely only on the tests (identifying symptoms) instead of addressing the root cause of the problem (using duplicate code).

Keep Unit Interfaces Small

Bunches of data that hang around together really ought to be made into their own object.
—Martin Fowler

Guideline:

- Limit the number of parameters per unit to at most 4.
- Do this by extracting parameters into objects.
- This improves maintainability because keeping the number of parameters low makes units easier to understand and reuse

UNIVERSITAS INDONESIA
FACULTY OF COMPUTER SCIENCE

There are many situations in the daily life of a programmer where long parameter lists seem unavoidable. In the rush of getting things done, you might add a few parameters more to that one method in order to make it work for exceptional cases. In the long term, however, such a way of working will lead to methods that are hard to maintain and hard to reuse. To keep your code maintainable it is essential to avoid long parameter lists, or unit interfaces, by limiting the number of parameters they have.

Why We Keep Unit Interfaces Small?

The reason are:

- Small Interfaces Are Easier to Understand and Reuse
- Methods with Small Interfaces Are Easier to Modify

FACULTY OF COMPUTER SCIENCE

Small Interfaces Are Easier to Understand and Reuse

As the codebase grows, the core classes become the API upon which a lot of other code in the system builds.

Methods with Small Interfaces Are Easier to Modify

Large interfaces do not only make your methods obscure, but in many cases also indicate multiple responsibilities (especially when you feel that you really cannot group your objects together anymore). In this sense, interface size correlates with unit size and unit complexity. So it is pretty obvious that methods with large interfaces are hard to modify.

Common Objections to Keeping Unit Interfaces Small

- Parameter Objects with Large Interfaces
  "The parameter object I introduced now has a constructor with too many parameters."
- Refactoring Large Interfaces Does Not Improve My Situation
  "When I refactor my method, I am still passing a lot of parameters to another method."
- Frameworks or Libraries Prescribe Interfaces with Long Parameter Lists
  "The interface of a framework we're using has nine parameters. How can I implement this interface without creating a unit interface violation?"

UNIVERSITAS INDONESIA
FACULTY OF COMPUTER SCIENCE

Parameter Objects with Large Interfaces

"The parameter object I introduced now has a constructor with too many parameters."

If all went well, you have grouped a number of parameters into an object during the refactoring of a method with a large interface. It may be the case that your object now has a lot of parameters because they apparently fit together. This usually means that there is a finer distinction possible inside the object.

Refactoring Large Interfaces Does Not Improve My Situation

"When I refactor my method, I am still passing a lot of parameters to another method."

Getting rid of large interfaces is not always easy. It usually takes more than refactoring one method. Normally, you should continue splitting responsibilities in your methods, so that you access the most primitive parameters only when you need to manipulate them separately.

Frameworks or Libraries Prescribe Interfaces with Long Parameter Lists

"The interface of a framework we're using has nine parameters. How can I implement this interface without creating a unit interface violation?"

Sometimes frameworks/libraries define interfaces or classes with methods that have long parameter lists. Implementing or overriding these methods will inevitably lead to long parameter lists in your own code.

Separate Concerns in Modules

In a system that is both complex and tightly coupled, accidents are inevitable.
—Charles Perrow's Normal Accidents theory in one sentence

Guideline:

- Avoid large modules in order to achieve loose coupling between them.
- Do this by assigning responsibilities to separate modules and hiding implementation details behind interfaces.
- This improves maintainability because changes in a loosely coupled codebase are much easier to oversee and execute than changes in a tightly coupled codebase.

FACULTY OF COMPUTER SCIENCE

Two principles are necessary to understand the significance of coupling between classes.

• Coupling is an issue on the class level of source code.

• Tight and loose coupling are a matter of degree. The actual maintenance consequence of tight coupling is determined by the number of calls to that class and the size of that class. Therefore, the more calls to a particular class that is tightly coupled, the smaller its size should be. Consider that even when classes are split up, the number of calls may not necessarily be lower. However, the coupling is then lower, because less code is coupled.

Why Separate Concerns in Modules?

The motivation are:

- Allow Developers to Work on Isolated Parts of the Codebase
- Ease Navigation Through the Codebase
- Prevent No-Go Areas for New Developers

FACULTY OF COMPUTER SCIENCE

Small, Loosely Coupled Modules Allow Developers to Work on Isolated Parts of the Codebase

When a class is tightly coupled with other classes, changes to the implementation of the class tend to create ripple effects through the codebase. For example, changing the interface of a public method leads to code changes everywhere the method is called. Besides the increased development effort, this also increases the risk that class modifications lead to bugs or inconsistencies in remote parts of the codebase.

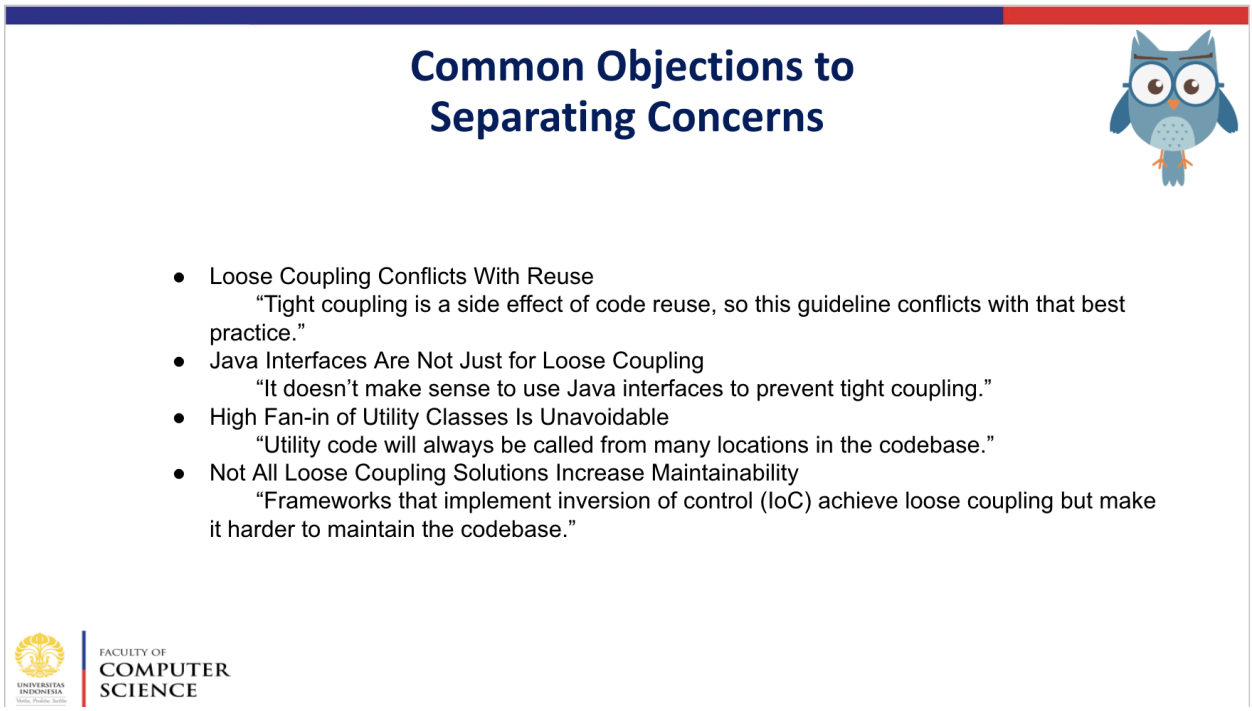
Small, Loosely Coupled Modules Ease Navigation Through the Codebase

Not only does a good separation of concerns keep the codebase flexible to facilitate future changes, it also improves the analyzability of the codebase since classes encap- sulate data and implement logic to perform a single task. Just as it is easier to name methods that only do one thing, classes also become easier to name and understand when they have one responsibility. Making sure classes have only one responsibility is also known as the single responsibility principle.

Small, Loosely Coupled Modules Prevent No-Go Areas for New Developers

Classes that violate the single responsibility principle become tightly coupled and accumulate a lot of code over time. As with the UserService example in the introduction of this chapter, these classes become intimidating to less experienced developers, and even experienced developers

are hesitant to make changes to their implementation. A codebase that has a large number of classes that lack a good separation of concerns is very difficult to adapt to new requirements.



Loose Coupling Conflicts With Reuse

"Tight coupling is a side effect of code reuse, so this guideline conflicts with that best practice."

Java Interfaces Are Not Just for Loose Coupling

"It doesn't make sense to use Java interfaces to prevent tight coupling."

High Fan-in of Utility Classes Is Unavoidable

"Utility code will always be called from many locations in the codebase."

Not All Loose Coupling Solutions Increase Maintainability

"Frameworks that implement inversion of control (IoC) achieve loose coupling but make it harder to maintain the codebase."

Couple Architecture Components Loosely

There are two ways of constructing a software design: one way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies.

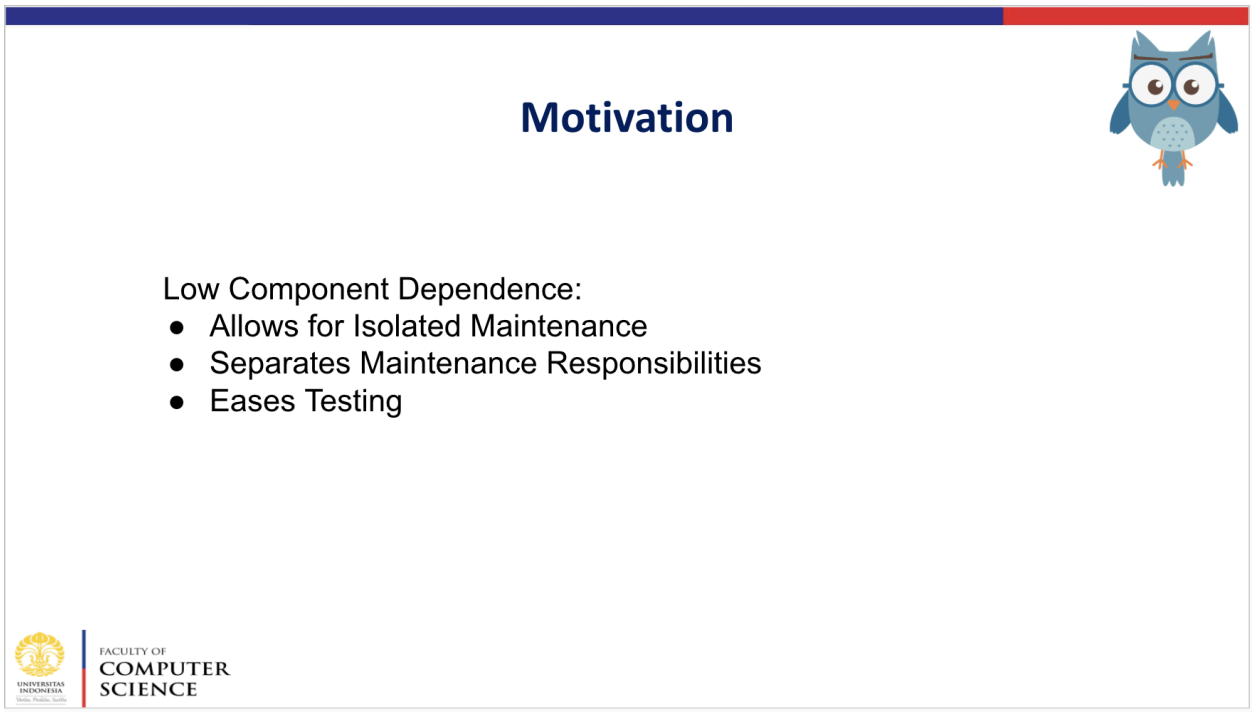
—C.A.R. Hoare

Guideline:

- Achieve loose coupling between top-level components.
- Do this by minimizing the relative amount of code within modules that is exposed to (i.e., can receive calls from) modules in other components.
- This improves maintainability because independent components ease isolated maintenance.

FACULTY OF COMPUTER SCIENCE

Having a clear view on software architecture is essential when you are building and maintaining software. A good software architecture gives you insight into what the system does, how the system does it, and how functionality is organized (in component groupings, that is). It shows you the high-level structure, the “skeleton” so to speak, of the system. Having a good architecture makes it easier to find the source code that you are looking for and to understand how (high-level) components interact with other components.



Low Component Dependence Allows for Isolated Maintenance

A low level of dependence means that changes can be made in an isolated manner. This applies when most of a component's code volume is either internal or outgoing. Isolated maintenance means less work, as coding changes do not have effects outside the functionality that you are modifying.

Low Component Dependence Separates Maintenance Responsibilities

If all components are independent from each other, it is easier to distribute responsibilities for maintenance among separate teams. This follows from the advantage of isolated modification. Isolation is in fact a prerequisite for efficient division of development work among team members or among different teams.

Low Component Dependence Eases Testing

Code that has a low dependence on other components (modules with mainly internal and outgoing code) is easier to test. For internal calls, functionality can be traced and tested within the component. For outgoing calls, you do not need to mock or stub functionality that is provided by other components (given that functionality in that other component is finished).

Common Objections to Loose Component Coupling

- Component Dependence Cannot Be Fixed Because the Components Are Entangled
  "We cannot get component dependence right because of mutual dependencies between components."
- No Time to Fix
  "In the maintenance team, we understand the importance of achieving low component dependence, but we are not granted time to fix it."
- Throughput Is a Requirement
  "We have a requirement for a so€ware architecture for a layer that puts through calls."

FACULTY OF COMPUTER SCIENCE

Component Dependence Cannot Be Fixed Because the Components Are Entangled

"We cannot get component dependence right because of mutual dependencies between components."

Entangled components are a problem that you experience most clearly during maintenance. When you achieve clearer boundaries for component responsibilities, it improves the analyzability and testability of the modules within. For example, modules with an extraordinary number of incoming calls may signal that they have multiple responsibilities and can be split up. When they are split up, the code becomes easier to analyze and test.

No Time to Fix

"In the maintenance team, we understand the importance of achieving low component dependence, but we are not granted time to fix it."

Development deadlines are real, and there may not be time for refactoring, or what a manager may see as "technical aesthetics." What is important is the trade-off. One should resolve issues that pose a real problem for maintainability. So dependencies should be resolved if the team finds that they inhibit testing, analysis, or stability.

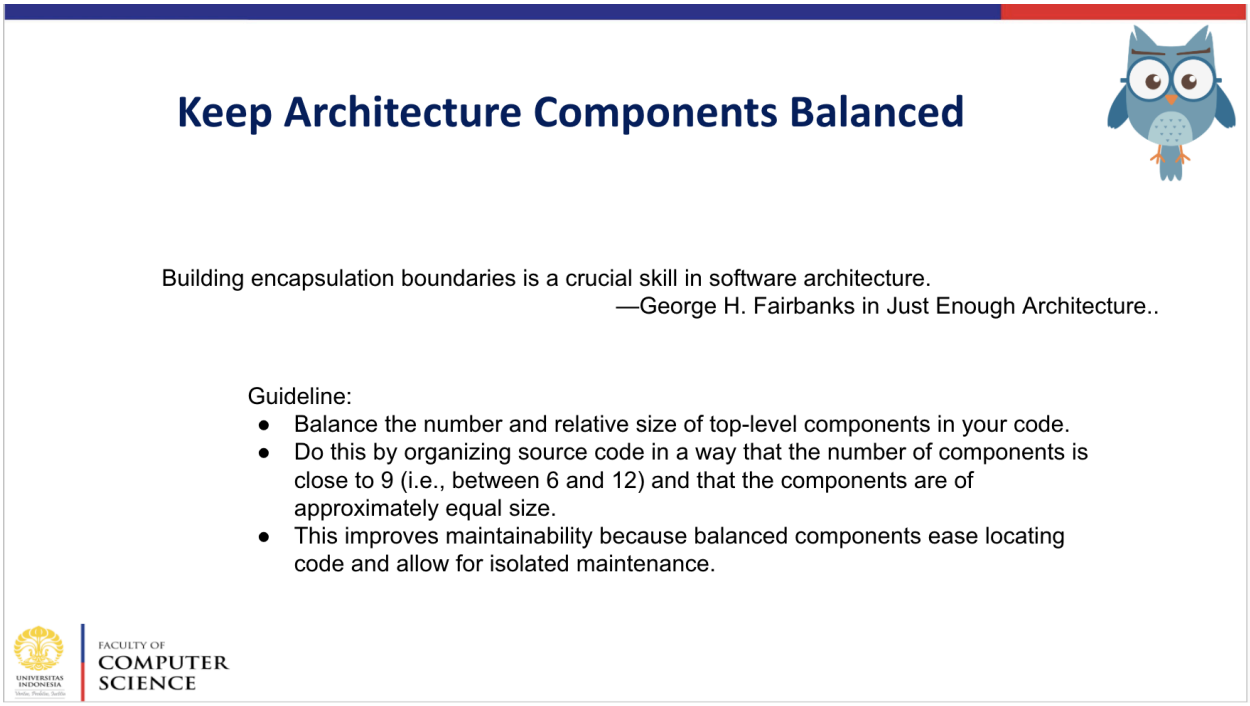
Throughput Is a Requirement

“We have a requirement for a software architecture for a layer that puts through calls.”

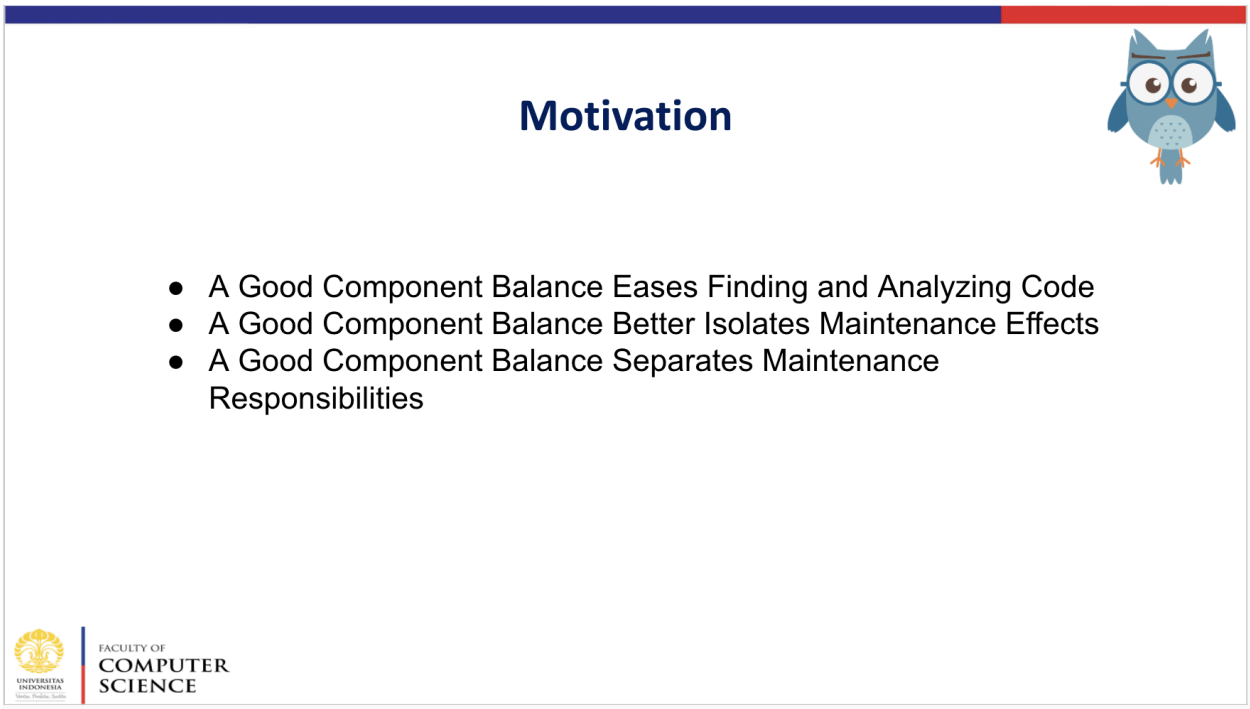
It is true that some architectures are designed to include an intermediate layer. Typically, this is a service layer that collects requests from one side (e.g., the user interface) and bundles them for passing on to another layer in the system. The existence of such a layer is not necessarily a problem—given that this layer implements loose coupling. It should have a clear separation of incoming and outgoing requests. So the module that receives requests in this layer:

• Should not process the request itself.

• Should not know where and how to process that request (its implementation details).



A well-balanced software architecture is one with not too many and not too few components, with sizes that are approximately equal. The architecture then has a good component balance. An example of component imbalance would be having a few very large components that contain a disproportionate amount of system logic and many small ones that dwindle in comparison.



A Good Component Balance Eases Finding and Analyzing Code

A clear code organization in components makes it easier to find the piece of code that you want to change. Of course, proper code hygiene helps in this process as well, such as using a consistent naming convention.

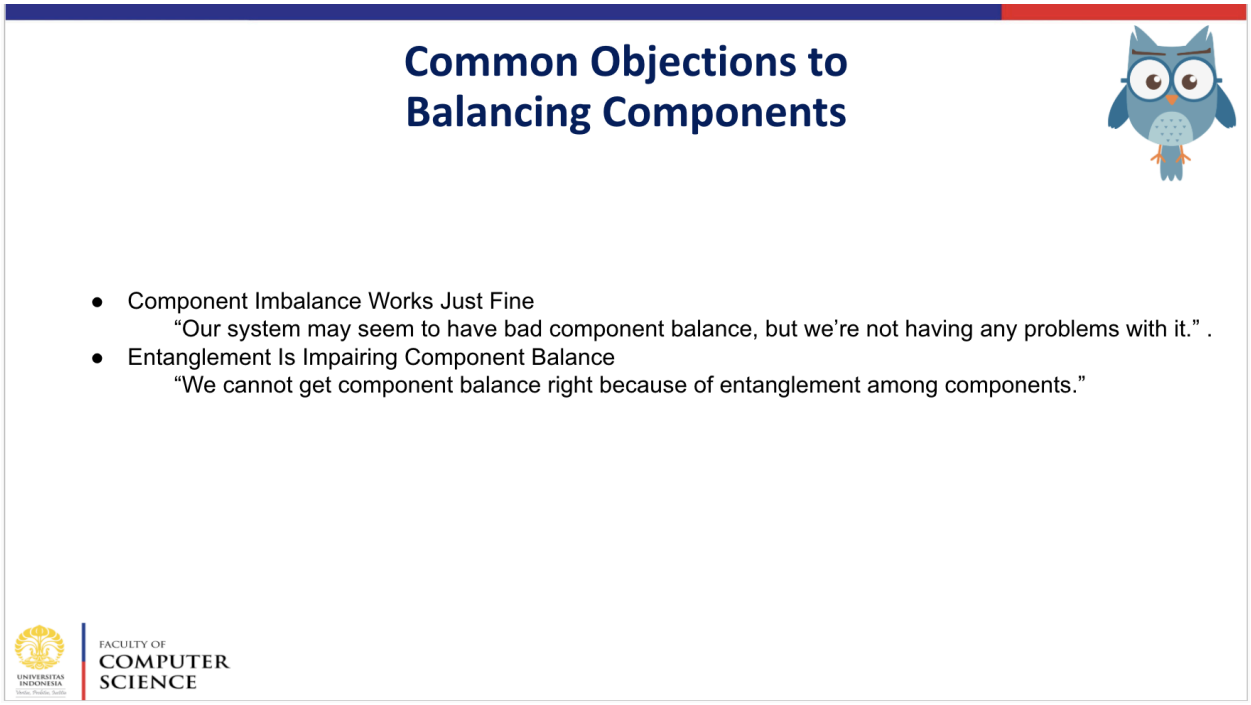
A Good Component Balance Better Isolates Maintenance Effects

When a system's component balance clearly describes functional boundaries, it has a proper separation of concerns, which makes for isolated behavior in the system. Isolated behavior within system components is relevant because it guards against unexpected effects, such as regression.

A Good Component Balance Separates Maintenance Responsibilities

Having clear functional boundaries between components makes it easier to distribute responsibilities for maintenance among separate teams. The number of components of a system and their relative size should indicate the system's decomposition into functional groups.

When a system has too many or too few components, it is considered more difficult to understand and harder to maintain. If the number of components is too low, it does not help you much to navigate through the functionalities of the system. On the other hand, too many components make it hard to get a clear overview of the entire system.



Component Imbalance Works Just Fine

"Our system may seem to have bad component balance, but we're not having any problems with it."

Component balance, as we define it, is not binary. There are different degrees of balance, and its definition allows for some deviation from the "ideal" of nine components of equal size. Whether a component imbalance is an actual maintenance burden depends on the degree of deviation, the experience of the maintenance team, and the cause of the imbalance.

Entanglement Is Impairing Component Balance

"We cannot get component balance right because of entanglement among components."

This situation points to another problem: technical dependence between components. Entanglement between components signals an improper separation of concerns.

Keep Your Codebase Small

Program complexity grows until it exceeds the capability of the programmer who must maintain it.

—7th Law of Computer Programming

Guideline:

- Keep your codebase as small as feasible.
- Do this by avoiding codebase growth and actively reducing system size.
- This improves maintainability because having a small product, project, and team is a success factor

FACULTY OF COMPUTER SCIENCE

A codebase is a collection of source code that is stored in one repository, can be compiled and deployed independently, and is maintained by one team. A system has at least one codebase. Larger systems sometimes have more than one codebase. A typical example is packaged software.

Motivation

- A Project That Sets Out to Build a Large Codebase Is More Likely to Fail
- Large Codebases Are Harder to Maintain
- Large Systems Have Higher Defect Density

FACULTY OF COMPUTER SCIENCE

Projects that aim to build a large codebase are likely to fail

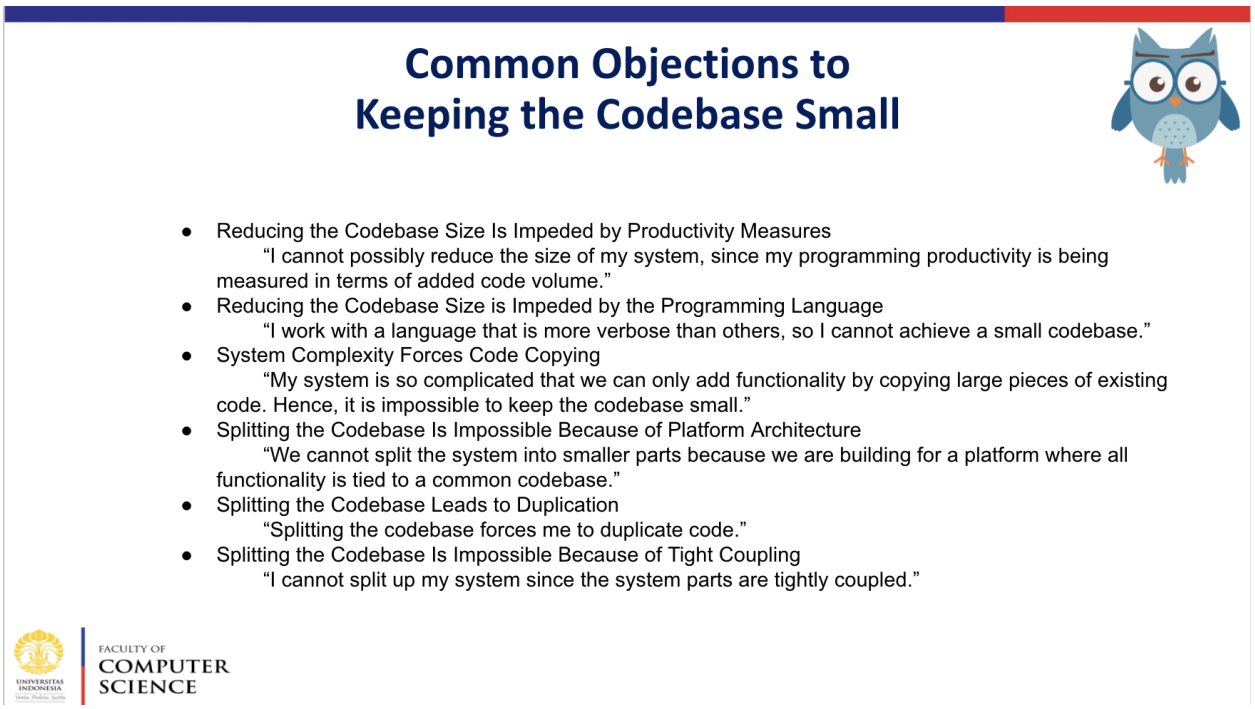
There is a strong correlation between project size and project risk. A large project results in a larger team, complex design, and longer project duration. As a result, there is more complex communication and coordination among stakeholders and team members, less overview of the software design, and more requirements that change over the course of the project.

Large Code Bases Are Harder to Maintain

The report explains that it is difficult to maintain large amounts of code

Large Systems Have a Higher Defect Density

The report also explains that large systems have High Defect Density



Reducing the Codebase Size Is Impeded by Productivity Measures

"I cannot possibly reduce the size of my system, since my programming productivity is being measured in terms of added code volume."

Reducing the Codebase Size is Impeded by the Programming Language

"I work with a language that is more verbose than others, so I cannot achieve a small codebase."

System Complexity Forces Code Copying

"My system is so complicated that we can only add functionality by copying large pieces of existing code. Hence, it is impossible to keep the codebase small."

Splitting the Codebase Is Impossible Because of Platform Architecture

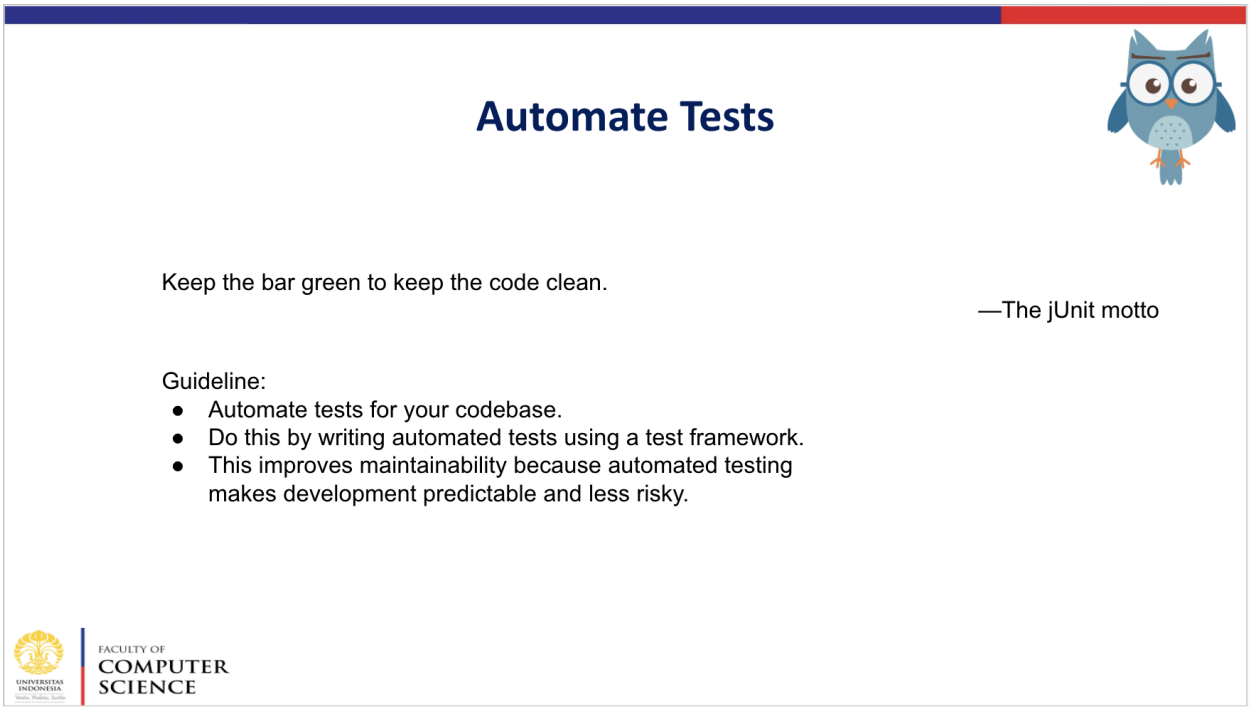
"We cannot split the system into smaller parts because we are building for a platform where all functionality is tied to a common codebase."

Splitting the Codebase Leads to Duplication

"Splitting the codebase forces me to duplicate code."

Splitting the Codebase Is Impossible Because of Tight Coupling

"I cannot split up my system since the system parts are tightly coupled."



Why Automating Tests

- Automated Testing Makes Testing Repeatable
- Automated Testing Makes Development Efficient
- Automated Testing Makes Code Predictable
- Tests Document the Code That Is Tested
- Writing Tests Make You Write Better Code

FACULTY OF COMPUTER SCIENCE

Automated Testing Makes Testing Repeatable

Just like other programs and scripts, automated tests are executed in exactly the same way every time they are run. This makes testing repeatable: if a certain test executes at two different points in time yet gives different answers, it cannot be that the test execution itself was faulty.

Automated Testing Makes Development Efficient

Automated tests can be executed with much less effort than manual tests. The effort they require is negligible and can be repeated as often as you see fit. They are also faster than manual code review. You should also test as early in the development process as possible, to limit the effort it takes to fix problems.

Automated Testing Makes Code Predictable

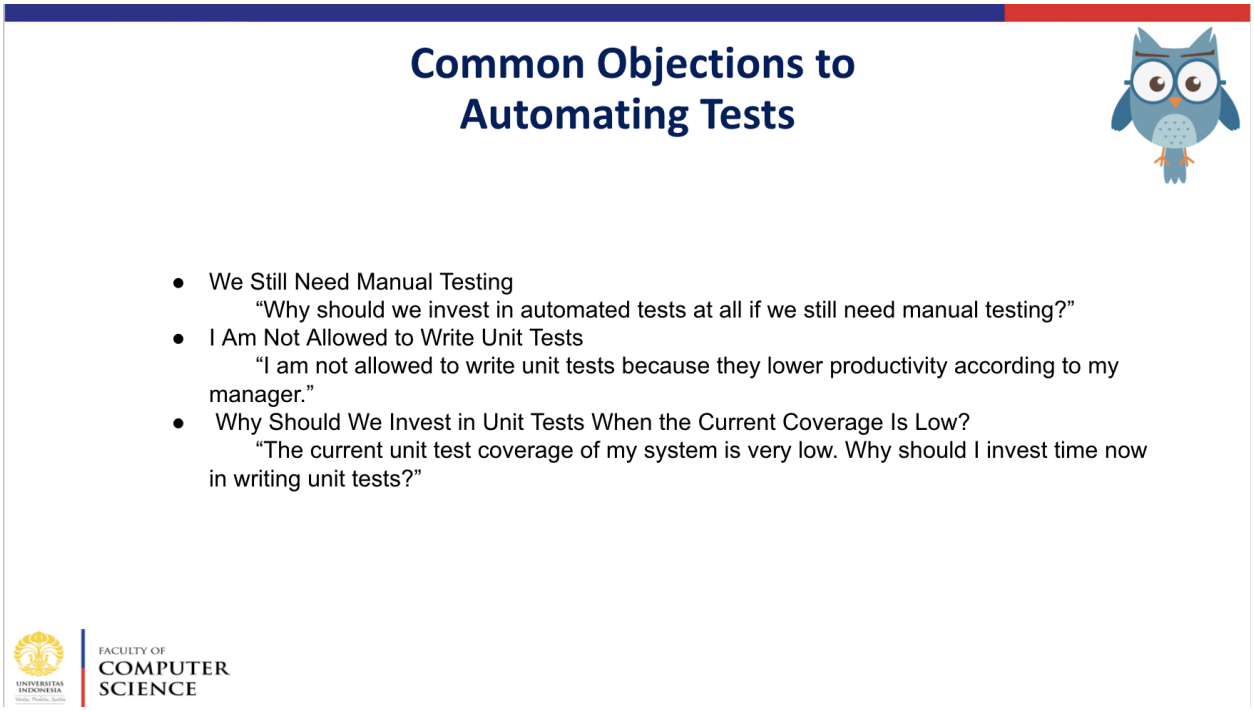
Technical tests can be automated to a high degree. Take unit tests and integration tests: they test the technical inner workings of code and the cohesion/integration of that code. Without being sure of the inner workings of your system, you might get the right results by accident.

Tests Document the Code That Is Tested

The script or program code of a test contains assertions about the expected behavior of the system under test.

Writing Tests Make You Write Better Code

Writing tests helps you to write testable code. As a side effect, this leads to code consisting of units that are shorter, are simpler, have fewer parameters, and are more loosely coupled.



We Still Need Manual Testing

"Why should we invest in automated tests at all if we still need manual testing?"

The answer to this question is simply because test automation frees up time to manually test those things that cannot be automated.

Consider the downsides of the alternative to automated tests. Manual testing has clear limitations. It is slow, expensive, and hard to repeat in a consistent manner. In fact, technical verification of the system needs to take place anyway, since you cannot manually test code that does not work. Because manual tests are not easily repeatable, even with small code changes a full retest may be needed to be sure that the system works as intended.

I Am Not Allowed to Write Unit Tests

"I am not allowed to write unit tests because they lower productivity according to my manager."

Writing unit tests during development actually improves productivity. It improves system code by shifting the focus from “what code should do” toward “what it should not do.” If you never take into account how the code may fail, you cannot be sure whether your code is resilient to unexpected situations.

The disadvantages of not having unit tests are mainly in uncertainty and rework. Every time a piece of code is changed, it requires a painstaking review to verify whether the code does what it is supposed to do.

Why Should We Invest in Unit Tests When the Current Coverage Is Low?

“The current unit test coverage of my system is very low. Why should I invest time now in writing unit tests?”

We have elaborated on the reasons why unit tests are useful and help you develop code that works predictably. However, when a very large system has little to no unit test code, this may be a burden. After all, it would be a significant investment to start writing unit tests from scratch for an existing system because you would need to analyze all units again. Therefore, you should make a significant investment in unit tests only if the added certainty is worth the effort. This especially applies to critical, central functionality and when there is reason to believe that units are behaving in an unintended manner. Otherwise, add unit tests incrementally each time you change existing code or add new code.

Write Clean Code

Writing clean code is what you must do in order to call yourself a professional.
—Robert C. Martin..

Guideline:
- Write clean code.
- Do this by not leaving code smells behind after development work.
- This improves maintainability because clean code is maintainable code

FACULTY OF COMPUTER SCIENCE

Code smells are coding patterns that hint that a problem is present. Introducing or not removing such patterns is bad practice, as they decrease the maintainability of code. Now, we discuss guidelines for keeping the codebase clean from code smells to achieve a “hygienic environment.”

Common Objections to Writing Clean Code

- Comments Are Our Documentation
  “We use comments to document the workings of the code.”
- Exception Handling Causes Code Additions
  “Implementing exception classes forces me to add a lot of extra code without visible benefits.”
- Why Only These Coding Guidelines?
  “We use a much longer list of coding conventions and quality checks in our team. This list of seven seems like an arbitrary selection with many important omissions.”

FACULTY OF COMPUTER SCIENCE

Comments Are Our Documentation

“We use comments to document the workings of the code.”

Comments that tell the truth can be a valuable aid to less experienced developers who want to understand how a piece of code works. In practice, however, most comments in code lie—not on purpose, of course, but comments often tell an outdated version of the truth. Outdated comments become more and more common as the system gets older. Keeping comments in sync with code is a task that requires precision and a lot of times is overlooked during maintenance.

Exception Handling Causes Code Additions

“Implementing exception classes forces me to add a lot of extra code without visible benefits.”

Exception handling is an important part of defensive programming: coding to prevent unstable situations and unpredictable system behavior. Anticipating unstable situations means trying to foresee what can go wrong. This does indeed add to the burden of analysis and coding. However, this is an investment. The benefits of exception handling may not be visible now, but they definitely will prove valuable in preventing and absorbing unstable situations in the future.

By defining exceptions, you are documenting and safeguarding your assumptions. They can later be adjusted when circumstances change.

Why Only These Coding Guidelines?

“We use a much longer list of coding conventions and quality checks in our team. This list of seven seems like an arbitrary selection with many important omissions.”

Having more guidelines and checks than the seven in this chapter is of course not a problem. These seven rules are the ones we consider the most important for writing maintainable code and the ones that should be adhered to by every member on the development team.

Misunderstanding: Maintainability Is Language-Dependent

"Our system uses a state-of-the-art programming language. Therefore, it is at least as maintainable as any other system."

FACULTY OF COMPUTER SCIENCE

SIG suggests that the choice of technology, specifically the programming language, is not the primary factor influencing maintainability. Their dataset includes Java systems that range from highly maintainable to poorly maintainable. The average maintainability of all Java systems in their benchmark is moderate, as is the case for systems developed in C#. This indicates that while it's feasible to develop highly maintainable systems using Java (and C#), the selection of either language doesn't ensure a system's maintainability. Evidently, there are other factors at play that dictate maintainability.

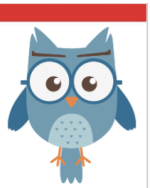
Misunderstanding: Maintainability Is Industry-Dependent

"My team makes embedded software for the car industry. Maintainability is different there."

FACULTY OF COMPUTER SCIENCE

The principles are universally applicable across various types of software development, encompassing embedded systems, gaming applications, scientific computing, as well as software components like compilers and database engines, and administrative tools. Although there are distinctions among these domains, such as the specialized programming languages often employed in scientific software, such as R for statistical analysis, it remains advisable in R to keep code units brief and straightforward. Embedded software, on the other hand, must function within environments where performance predictability is paramount, and resources are limited. Consequently, in situations where a trade-off between performance and maintainability is necessary, performance takes precedence over maintainability. However, regardless of the specific domain, the characteristics delineated in ISO 25010 remain pertinent and applicable.

Misunderstanding: Maintainability Is the Same as the Absence of Bugs



"You said the system has above-average maintainability. However, it turns out it is full of bugs!"

FACULTY OF COMPUTER SCIENCE

As per ISO 25010 definitions, a system could exhibit high maintainability while potentially lacking in other quality aspects. Therefore, it's possible for a system to demonstrate superior maintainability yet face challenges in areas such as functional suitability, performance, reliability, and others. Having above-average maintainability simply implies that the required modifications to minimize bugs can be carried out with high efficiency and effectiveness.

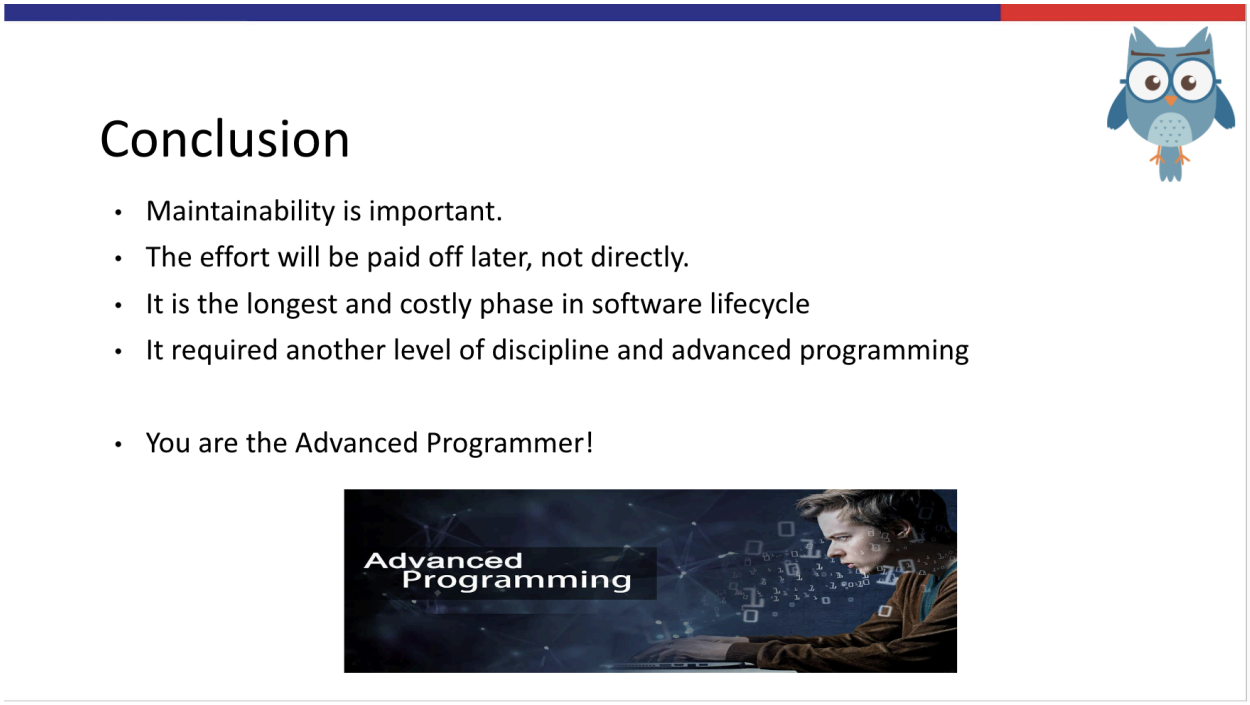
Misunderstanding: Maintainability Is a Binary Quantity

"My team repeatedly has been able to fix bugs in this system. Therefore, it has been proven that it is maintainable."

FACULTY OF COMPUTER SCIENCE

As per the definition outlined in ISO 25010, the maintainability of source code isn't a straightforward binary concept. Instead, it refers to the extent to which alterations can be executed efficiently and effectively. Thus, the pertinent inquiry isn't merely whether changes, such as bug fixes, were implemented, but rather, the level of effort expended in rectifying the issue (efficiency), and whether the correction was executed accurately (effectiveness).
In line with ISO 25010's definition of maintainability, it can be argued that a software system never achieves perfect maintainability nor does it fall into the category of perfect unmaintainability. In real-world scenarios, at SIG, we have encountered systems deemed as unmaintainable. These systems exhibited such a minimal level of efficiency and effectiveness in modification that the system owner couldn't viably sustain them.

Conclusion About Software Maintainability



This statement emphasizes the importance of maintainability in software development and highlights several key points:

Importance of Maintainability: Maintainability refers to the ease with which a software system can be maintained and updated over time. It's crucial because software often needs to be modified, fixed, or enhanced after it's initially developed. Without good maintainability, these tasks become difficult and costly.

Long-term Investment: The effort put into ensuring maintainability may not yield immediate benefits but pays off in the long run. While it might seem easier to prioritize features and

functionalities during the initial development phase, neglecting maintainability can lead to significant challenges and costs later on.

Longest and Costly Phase: The phase of maintaining software is depicted as the longest and most expensive phase in the software lifecycle. This underscores the idea that the majority of a software system's life is spent in maintenance rather than initial development. Therefore, investing in maintainability upfront can help mitigate these long-term costs.

Discipline and Advanced Programming: Achieving maintainability often requires a higher level of discipline and advanced programming practices. This includes writing clean, modular, and well-documented code, adhering to design principles, implementing proper error handling, and utilizing appropriate software architecture patterns. Such practices make it easier for developers to understand, modify, and debug the codebase as needed.

3. Tutorial & Exercise

Development

Open your Module 2 Exercise project using IntelliJ or your favorite editor.

- Create a new branch named **before-solid** based on the latest version of your `main`/`master` branch.

Model

Create a model with the name **Car.Java in the model package**. Create a model like the code snippet below.

```
package id.ac.ui.cs.advprog.eshop.model;
import lombok.Getter;
import lombok.Setter;
32 usages  new *
@Getter @Setter
public class Car {
    private String carId;
    private String carName;
    private String carColor;
    private int carQuantity;
}
```

- Commit **Car.java** to your Git repository

Repository

Create a repository with the name **CarRepository.Java** in the repository package. Create a repository like the code snippet below.

```java
package id.ac.ui.cs.advprog.eshop.repository;
import id.ac.ui.cs.advprog.eshop.model.Car;
import org.springframework.stereotype.Repository;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import java.util.UUID;
new *
@Repository
public class CarRepository {  Complexity is 7 It's time to do something...
    no usages
    static int id = 0;
    6 usages
    private List<Car> carData = new ArrayList<>();
    1 usage  new *
    public Car create(Car car){  Complexity is 4 Everything is cool!
        if(car.getCarId() == null){
            UUID uuid = UUID.randomUUID();
            car.setCarId(uuid.toString());
        }
        carData.add(car);
        return car;
    }
    1 usage  new *
    public Iterator<Car> findAll(){
        return carData.iterator();
    }
```

```java
    public Car findById(String id) {  Complexity is 5 Everything is cool!
        for (Car car : carData) {
            if (car.getCarId().equals(id)) {
                return car;
            }
        }
        return null;
    }
    1 usage   new *
    public Car update(String id, Car updatedCar) {  Complexity is 6 It's time to do something...
        for (int i = 0; i < carData.size(); i++) {
            Car car = carData.get(i);
            if (car.getCarId().equals(id)) {
                // Update the existing car with the new information
                car.setCarName(updatedCar.getCarName());
                car.setCarColor(updatedCar.getCarColor());
                car.setCarQuantity(updatedCar.getCarQuantity());
                return car;
            }
        }
        return null; // Handle the case where the car is not found
    }
    1 usage   new *
    public void delete(String id) { carData.removeIf(car -> car.getCarId().equals(id)); }

}
```

- Commit **CarRepository.java** to your Git repository

Service

Create a service with the name **CarService.Java** in the service package. Create a service like the code snippet below.

```java
package id.ac.ui.cs.advprog.eshop.service;
import id.ac.ui.cs.advprog.eshop.model.Car;
import java.util.List;

1 usage  1 implementation  new *
public interface CarService {
    1 usage  1 implementation  new *
    public Car create(Car car);
    1 usage  1 implementation  new *
    public List<Car> findAll();
    1 usage  1 implementation  new *
    Car findById(String carId);
    1 usage  1 implementation  new *
    public void update(String carId, Car car);
    1 usage  1 implementation  new *
    public void deleteCarById(String carId);
}
```

Add a service with the name **CarServiceImpl.Java** in the service package. Create a service like the code snippet below.

```java
package id.ac.ui.cs.advprog.eshop.service;
import id.ac.ui.cs.advprog.eshop.model.Car;
import id.ac.ui.cs.advprog.eshop.repository.CarRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

new *
@Service
public class CarServiceImpl implements CarService{ Complexity is 4 Everything is cool!
    @Autowired
    private CarRepository carRepository;
    1 usage  new *
    @Override
    public Car create(Car car) {
        // TODO Auto-generated method stub
        carRepository.create(car);
        return car;
    }
    1 usage  new *
    @Override
    public List<Car> findAll() { Complexity is 3 Everything is cool!
        Iterator<Car> carIterator = carRepository.findAll();
        List<Car> allCar = new ArrayList<>();
        carIterator.forEachRemaining(allCar::add);
        return allCar;
    }
```

```
    1 usage  new *
    @Override
    public Car findById(String carId){
        Car car = carRepository.findById(carId);
        return car;
    }
    1 usage  new *
    @Override
    public void update(String carId, Car car) {
        // TODO Auto-generated method stub
        carRepository.update(carId, car);
    }
    1 usage  new *
    @Override
    public void deleteCarById(String carId) {
        // TODO Auto-generated method stub
        carRepository.delete(carId);
    }
}
```

- Commit **CarService.java** and **CarServiceImpl.java** to your Git repository

Controller

Open your **ProductController.Java** and **add** the code snippet below.

```java
package id.ac.ui.cs.advprog.eshop.controller;
import id.ac.ui.cs.advprog.eshop.model.Car;
import id.ac.ui.cs.advprog.eshop.model.Product;
import id.ac.ui.cs.advprog.eshop.service.CarServiceImpl;
import id.ac.ui.cs.advprog.eshop.service.ProductService;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.*;
import java.util.List;

1 inheritor  nancenka *
@Controller
@RequestMapping("/product")
public class ProductController {  Complexity is 3 Everything is cool!
    @Autowired
    private ProductService service;

    nancenka
    @GetMapping("/create")
    public String createProductPage(Model model) {
        Product product = new Product();
        model.addAttribute( attributeName: "product", product);
        return "createProduct";
    }
    nancenka
    @PostMapping("/create")
    public String createProductPost(@ModelAttribute Product product, Model model){
        service.create(product);
        return "redirect:list";
    }
```

```java
    @GetMapping("/list")
    public String productListPage(Model model){
        List<Product> allProducts = service.findAll();
        model.addAttribute( attributeName: "products", allProducts);
        return "productList";
    }
    nancenka
    @GetMapping("/edit/{productId}")
    public String editProductPage(@PathVariable String productId, Model model) {
        Product product = service.findById(productId);
        model.addAttribute( attributeName: "product", product);
        return "editProduct";
    }
    nancenka *
    @PostMapping("/edit")
    public String editProductPost(@ModelAttribute Product product, Model model) {
        System.out.println(product.getProductId());
        service.update(product.getProductId(), product);

        return "redirect:list";
    }

    nancenka
    @PostMapping("/delete")
    public String deleteProduct(@RequestParam("productId") String productId) {
        service.deleteProductById(productId);
        return "redirect:list";
    }
}
new *
```

```java
@Controller
@RequestMapping("/car")
class CarController extends ProductController{ Complexity is 3 Everything is cool!
    @Autowired
    private CarServiceImpl carservice;

    new *
    @GetMapping("/createCar")
    public String createCarPage(Model model) {
        Car car = new Car();
        model.addAttribute( attributeName: "car", car);
        return "createCar";
    }

    new *
    @PostMapping("/createCar")
    public String createCarPost(@ModelAttribute Car car, Model model){
        carservice.create(car);
        return "redirect:listCar";
    }
    new *
    @GetMapping("/listCar")
    public String carListPage(Model model){
        List<Car> allCars = carservice.findAll();
        model.addAttribute( attributeName: "cars", allCars);
        return "carList";
    }
```

```
    new *
    @GetMapping("/editCar/{carId}")
    public String editCarPage(@PathVariable String carId, Model model) {
        Car car = carservice.findById(carId);
        model.addAttribute( attributeName: "car", car);
        return "editCar";
    }

    new *
    @PostMapping("/editCar")
    public String editCarPost(@ModelAttribute Car car, Model model) {
        System.out.println(car.getCarId());
        carservice.update(car.getCarId(), car);

        return "redirect:listCar";
    }

    new *
    @PostMapping("/deleteCar")
    public String deleteCar(@RequestParam("carId") String carId) {
        carservice.deleteCarById(carId);
        return "redirect:listCar";
    }
}
```

- Commit **ProductController.java** to your Git repository

Template

Copy and paste the following code to the template directory:

CreateCar.html: https://pastecode.io/s/f193v2xb
CarList.html: https://pastecode.io/s/47z8qw6w
EditCar.html: https://pastecode.io/s/0zgzz3ze

You may use your own HTML pages.

- Commit HTML templates to your Git repository

Run Application

1. Press Shift+F10 or push the “Run” (triangle button) on the upper right of IntelliJ to run your application.
2. Open http://localhost:8080/car/listCar

Exercise

- Push all your (committed) source code on **before-solid** branch!
- If everything goes well, merge **before-solid** branch to **main!**
- Create a new branch **after-solid** and apply SOLID Principle on your project**.**
- If everything goes well, merge **after-solid** branch to **main!**

Take a close look at your existing code.

- Have you implemented SRP? If yes, explain the principle; if not, please change your code as per SRP. Keep in mind that SRP is a class that has only one reason to change. In other words, a class should have only one responsibility or encapsulate only one aspect of the software's functionality.
- Have you implemented OCP? If yes, explain the principle, if not, please modify your code as per OCP. Remember that OCPs are software entities (classes, modules, functions, etc.) that should be open for development but closed for modification. This means you should be able to extend a module's behavior without changing its source code.
- Have you implemented LSP? If yes, explain the principle; if not, modify your code to suit the LSP. It should be remembered that the LSP is an object from the superclass that must be replaced with an object from the subclass without affecting the correctness of the program. In other words, subclasses must be replaceable with their base class without changing desired program properties, such as correctness and consistency.
- Have you implemented an ISP? If yes, explain the principle; if not please modify your code according to ISP. Keep in mind that ISPs recommend that large interfaces be broken down into smaller, more specific interfaces so that clients only need to know the methods that are relevant to them.
- Have you implemented DIP? If yes, explain the principle; if not change your code as per DIP. Note that DIP recommends that high-level modules do not depend on low-level modules. Both must rely on abstractions. Additionally, abstraction should not depend on details; details must rely on abstraction.
- Commit and Push your work to branch **after-solid**

Reflection

Apply the SOLID principles you have learned. You are allowed to modify the source code according to the principles you want to implement. Please answer the following questions:

1) Explain what principles you apply to your project!

2) Explain the advantages of applying SOLID principles to your project with examples.

3) Explain the disadvantages of not applying SOLID principles to your project with examples.

Please write the answer in the **README.md file.**

Grading Scheme

Scale

All components will be scored on scale 0-4. Grade 4 is the highest grade, equivalent to A. Score 0 is for not submitting.

Components

- 60% - Commits
- 20% - Reflection
- 20% - Correctness

Rubrics

	Score 4	Score 3	Score 2	Score 1
Correctness	Apply three or more principles correctly	Apply at least two principles correctly	Apply at least two principles correctly	Incorrect program
Reflection (for each question)	The description is sound.	The description is sound.	The description is not sound although still related.	The description is not sound. It is not related to the topics.
Commit	All requested commits are registered and correct.	At least 50% of the requested commits are registered and correct.	At least 25% of the requested commits are registered and correct.	Less than 25%.