

OVERVIEW Principles of Detailed Design II: Component Design, SRP, OCP and LSP

Architecture names the components; component design decides what goes inside them. This lecture shows how a component becomes classes and interfaces, refreshes the four object-oriented ideas you need for that (classes, abstract classes and interfaces, inheritance, polymorphism), and then covers the first three SOLID principles. Each principle is a test you can run on a class diagram.

Component design

Defines the internal structure (classes, data) and behavior (algorithms, interactions) of each architectural component. In OO it is drawn as class diagrams.

SRP: Single Responsibility

A class should have one reason to change, meaning one actor or policy drives its changes. It is about responsibilities, not about having one method.

OCP: Open-Closed

Add new behavior by adding new classes behind a stable abstraction, not by editing working code.

LSP: Liskov Substitution

A subclass must be usable anywhere its parent is expected without breaking what callers rely on.

HOW THE LECTURE IS ORGANIZED

Slides 1–6 Component design: from architecture components to classes and interfaces

Slides 7–14 OOP refresher: abstract classes, interfaces, inheritance, polymorphism (Java and UML)

Slides 15–21 SOLID intro and SRP: Person/Email, Employee, payroll exercise

Slides 22–31 OCP: the game engine redesign and the salary exercise

Slides 32–38 LSP: flying “terrestrial” character, square/rectangle, License, fruit exercise

Slide 39 What comes next: ISP, DIP and internal behavior

EXAM FOCUS

Confirmed: Final 252 Q5 (5 marks) asked you to name the SRP violation in a BookManager class and re-draw it, then draw an OCP class diagram so E-books can be added (see slides 21 and 31). LSP has not appeared on a past paper here, so the LSP questions on this page are predictions.

HOW TO READ THIS PAGE

Every original slide is shown unchanged. **Yellow marks** on a slide point to a numbered note: hover or tap a mark to jump to its note, and hover a note to see its marks. Questions tagged **Past exam** appeared on a real paper; **Prediction** questions are likely angles, not confirmed.



Software Design and Architecture

[Principles of Detailed Design] – Chapter 05, L02

NOTES

- 1 This is the second lecture of Chapter 5. Lecture 1 described the detailed design process; this one is about the **principles** that make a detailed design good: easy to change, reuse and test.

SLIDE 2 / 39 **Lecture Outlines**

Lecture Outlines

- **Overview of Component Design**
- **Designing Internal Structure of Components (OO Strategy)**
- **Design Principles for Internal Component Design (SOLID)**
 - ✓ Single Responsibility Principle
 - ✓ Open-Closed Principle
 - ✓ Liskov Substitution Principle
 - ✓ Interface-Segregation Principle
 - ✓ Dependency-Inversion Principle
- **Designing Internal Behavior of Components**

NOTES

- 1 **Internal structure** means which classes and interfaces a component contains and how they relate. It is drawn with class diagrams.
- 2 SOLID is five principles. Only the first three (S, O, L) are taught in this lecture; slide 39 moves I and D to the next one.
- 3 **Internal behavior** (what the classes do at run time, drawn with sequence or state diagrams) is also left for the next lecture.

SLIDE 3 / 39 Overview of Component Design

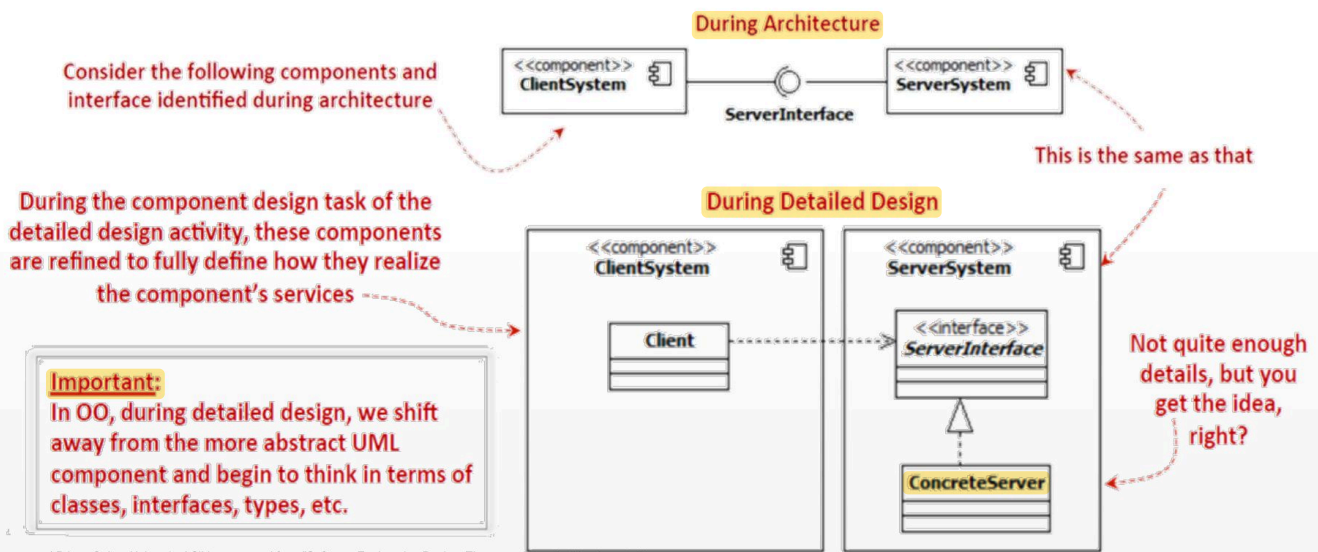
Overview of Component Design

- Component design (also known as component-level design) refers to the **detailed design** task of **defining the internal logical structure and behavior of components**.
 - ✓ Which components?
 - The components **identified during the architecture activity**.
- In object-oriented systems, the **internal structure** of components is typically modeled using UML through **one or more class diagrams**

NOTES

- 1 Component design is the part of detailed design that opens each component up. **Structure** is what it is made of (classes, attributes, interfaces); **behavior** is how those parts act to deliver the component's services.
- 2 It does not invent new top-level components. It refines the ones the architect already chose, so every class you design belongs to one architectural component.
- 3 In an object-oriented system each component's inside is usually shown with one or more class diagrams. A large component may need several.

Overview of Component Design



NOTES

- 1 How to read the top diagram (architecture view):
 - ClientSystem, ServerSystem**
UML components: boxes with the small component icon. They are black boxes at this stage.
 - Ball-and-socket on ServerInterface**
The circle (ball) is the interface `ServerSystem` **provides**; the half-circle (socket) is what `ClientSystem` **requires**. Joined, they mean the client uses that service.
- 2 The bottom diagram is the same two components after detailed design: they are opened up to show the classes inside.
 - Client**
A class inside `ClientSystem`.
 - <<interface>> ServerInterface**
The provided interface, now an actual interface type inside `ServerSystem`.
- 3 The dashed arrow with an open arrowhead from `Client` to `ServerInterface` is a **dependency**: `Client` calls the interface and never names the concrete class.
- 4 The dashed line with a hollow triangle from `ConcreteServer` to `ServerInterface` is **realization**: `ConcreteServer` implements the interface. Another class could replace it and `Client` would not change.
- 5 The takeaway box says it in one line: in detailed design you stop thinking in components and start thinking in classes, interfaces and types.

Overview of Component Design

- During **component design**, the **internal data structures**, **algorithms**, **interface details**, and **communication mechanisms** for all components are defined.
 - ✓ For this reason, component design provides the most significant mechanism for determining the **functional correctness** of the software system
 - If the requirement says “*The payment system must correctly calculate total cost including tax and discount*”, then functional correctness means that the designed components (e.g., **TaxCalculator**, **DiscountManager**, **PaymentProcessor**) must work together to always produce the right total, under all valid inputs.

NOTES

- 1 Four things are decided in component design. **Data structures** say how information is stored. **Algorithms** say how results are computed. **Interface details** are the exact method names, parameters and return types. **Communication mechanisms** say how components talk: direct calls, messages or events.
- 2 That makes component design the place where functional correctness is won or lost. Architecture can be perfect, but if the algorithm or the data passed between classes is wrong, the system still gives wrong answers.
- 3 In the payment example the three classes must agree on details like the order of operations (discount before tax?) and the number format. Only then is the total right for every valid input. Those agreements are exactly what component design writes down.

Overview of Component Design

- The work produced during component design contributes significantly to the functional success of the system.
- In OO, before we can become a component designers, we must understand the following concepts:
 - ✓ **Classes / objects**
 - ✓ **Abstract classes and Interfaces**
 - ✓ **Inheritance**
 - ✓ **Polymorphism**

NOTES

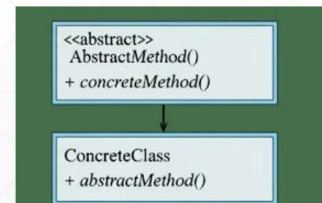
- 1 Because component design fixes the actual logic, it is a major factor in whether the system does what users need.
- 2 The four object-oriented ideas a component designer must be fluent in. Slides 7–14 refresh each one:
 - Class / object**
A class is the blueprint; an object is one instance created from it at run time.
 - Abstract class / interface**
Types that define a contract other classes must complete or implement.
 - Inheritance**
A subclass reuses and specializes a parent ("is-a").
 - Polymorphism**
One call, such as `animal.sound()`, runs different code depending on the actual object.

What Is Abstract Class?

- A class which has the **abstract keyword** in its declaration is called abstract class. Abstract classes should have **at least one abstract method**, i.e., methods without a body. It can have multiple concrete methods.
- Abstract classes allow you to create blueprints for concrete classes. But the **inheriting class should implement the abstract method**.
- Abstract classes **cannot be instantiated**.

Important Reasons For Using Abstract Class

- ✓ Abstract classes offer **default functionality** for the subclasses.
- ✓ Provides a template for future specific classes
- ✓ Helps you to define a common interface for its subclasses
- ✓ Abstract class allows **code reusability**.



NOTES

- 1 An abstract class is declared with the **abstract** keyword. It can mix **abstract methods** (a signature with no body) and **concrete methods** (normal methods with code). The slide's definition requires at least one abstract method. Java itself also accepts an abstract class with none, but answer with the slide's definition in the exam.
- 2 Any concrete (non-abstract) subclass must give a body to every abstract method it inherits. If it doesn't, it must be declared abstract too.
- 3 "Cannot be instantiated" means `new Animal()` is a compile error when `Animal` is abstract. You can still declare a variable of that type: `Animal a = new Cat();`.
- 4 The four reasons all come down to sharing: common code lives once in the parent (default functionality, reuse), and the abstract methods fix a common set of operations every subclass must offer (template, common interface).
- 5 Reading the small diagram:
 - Top box «abstract»**
The abstract class. It holds an abstract method (*AbstractMethod()*) and `+ concreteMethod()`, which has a body.
 - Arrow to ConcreteClass**
`ConcreteClass` derives from it and supplies `+ abstractMethod()`. The slide draws the arrow downward; standard UML points a hollow triangle from the subclass up to the parent.

Prediction

Q1

True or false: an abstract class can contain concrete methods, and it can be instantiated with `new`.

Answer: Half true. It can contain concrete methods, but it can never be instantiated directly. Only a concrete subclass can be created.

SLIDE 8 / 39 What is Interface?

What is Interface?

- A special type of abstract classes
- The interface is a blueprint that can be used to implement a class. The interface does not contain any concrete methods (methods that have code). All the methods of an interface are abstract methods.
- An interface cannot be instantiated. However, classes that implement interfaces can be instantiated. Interfaces never contain instance variables but, they can contain public static final variables (i.e., constant class variables)
 - ✓ Interfaces are used to achieve abstraction.
 - ✓ Designed to support dynamic method resolution at run time
 - ✓ It helps you to achieve loose coupling.
 - ✓ Allows you to separate the definition of a method from the inheritance hierarchy

NOTES

- 1 An interface is a pure contract: it lists method signatures and no code. The slide says no concrete methods at all. Modern Java also allows `default` and `static` methods, but for this course an interface has only abstract methods.
- 2 You cannot write `new AnimalInterface()`. You create an object of a class that implements it and refer to it through the interface type.
- 3 Interfaces hold no instance variables (no per-object state). The only fields allowed are constants: `public static final`.
- 4 Benefits: **loose coupling**, because callers depend on the contract and not on a specific class. **Dynamic method resolution**: at run time Java runs the method of the actual object behind the interface. It also lets unrelated classes share a capability outside one inheritance tree, since a Java class can implement many interfaces but extend only one class.

Prediction

Q1

Give two differences between an abstract class and an interface (as defined in this course).

Answer: An abstract class can have concrete methods and instance variables; an interface has only abstract methods and constants. A class extends one abstract class (`extends`) but can implement many interfaces (`implements`). In UML an abstract class is shown in italics, an interface with «interface».

SLIDE 9 / 39 Further reading

- <https://www.guru99.com/interface-vs-abstract-class-java.html>
- <https://www.guru99.com/uml-relationships-with-example.html#4>

NOTES

- 1 Further reading on when to choose an interface or an abstract class in Java.
- 2 Further reading on UML relationships. The #4 anchor jumps to the section on generalization and realization, the two arrows used on slides 10–12.

Refresh your memory ... OOP concepts (Interface)

```
public interface AnimalInterface {
    public void sound();
    public void drawe ();
}
```

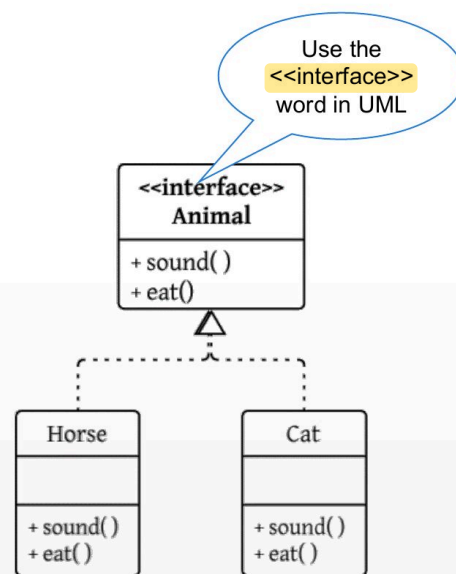
Use the **implements** reserved word in Java

```
public class Cat implements AnimalInterface{

    public void sound(){
        System.out.println("Miaw");
    }

    public void eat(){
        System.out.println("Cat is eating fish!");
    }
}
```

Prince Sultan University | Slides prepared from "Software Engineering Design: Theory and Practice" book and slides



10

NOTES

- The code, line by line:

CODE	WHAT IT DOES
public interface AnimalInterface {	Declares a contract named AnimalInterface.
public void sound();	A required method: signature only, ending in <code>;</code> , with no body.
public void drawe ();	A second required method (spelled "drawe" on the slide).
}	End of the contract.
public class Cat implements AnimalInterface{	implements promises that Cat supplies a body for every method in the interface.
public void sound(){ System.out.println("Miaw"); }	Cat's own version of sound(): prints Miaw.
public void eat(){ System.out.println("Cat is eating fish!"); }	Cat's eat() body.

- Watch the mismatch:** the interface asks for `drawe()`, but Cat provides `eat()`. As printed, Java would reject Cat ("does not override abstract method drawe()"). The UML on the right shows what was intended: the interface has `sound()` and `eat()`.

3 Reading the UML:

«interface» Animal

The stereotype in guillemets marks the box as an interface.

Horse, Cat

Classes that each provide `+ sound()` and `+ eat()`.

Dashed lines with a hollow triangle

Realization: the triangle points at the interface each class implements.

EXAM ANGLE

Prediction

Q1

Which UML notation shows that class Cat implements interface Animal?

Answer: A dashed line from Cat to Animal ending in a hollow triangle at Animal (realization), with Animal marked «interface».

SLIDE 11 / 39 Refresh your memory ... OOP concepts (Abstract Classes)

Refresh your memory ... OOP concepts (Abstract Classes)

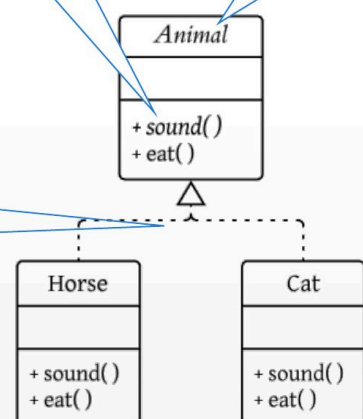
```
public abstract class Animal {  
  
    public abstract void sound();  
  
    public void eat() {  
        System.out.println("Animal is eating..");  
    }  
}  
  
public class Cat extends Animal {  
  
    public void sound() {  
        System.out.println("Miaw");  
    }  
  
    public void eat() {  
        System.out.println("Cat is eating fish!");  
    }  
}
```

Use the **extends** reserved word in Java

Abstract method is italic in UML

Abstract class is italic in UML

Realization Relationship



NOTES

- 1 The code, line by line:

CODE	WHAT IT DOES
<pre>public abstract class Animal {</pre>	An abstract class: it can't be instantiated.
<pre> public abstract void sound();</pre>	Abstract method: no body, and every concrete subclass must implement it.
<pre> public void eat() { System.out.println("Animal is eating.."); } }</pre>	Concrete method: inherited as-is unless a subclass overrides it.
<pre>public class Cat extends Animal {</pre>	<code>extends</code> = inherits from Animal (a Java class can extend only one class).
<pre> public void sound(){ System.out.println("Miaw"); }</pre>	Required: fills in the abstract <code>sound()</code> .
<pre> public void eat(){ System.out.println("Cat is eating fish!"); }</pre>	Optional: overrides the inherited <code>eat()</code> with Cat's version.

- 2 In UML, italics carry meaning: the class name *Animal* in italics means abstract class, and *sound()* in italics means abstract operation. `eat()` is upright because it has a body.
- 3 **Notation trap:** the slide calls the dashed arrows "Realization". Standard UML uses realization (dashed line, hollow triangle) for implementing an **interface**, and generalization (solid line, hollow triangle) for **extending a class**, abstract or not. In an exam, draw `extends` with a solid line and say why.

EXAM ANGLE

Prediction

Q1

A class Dog extends an abstract class Animal. Which UML line connects them, and how is Animal's name written?

Answer: A solid line with a hollow triangle pointing at Animal (generalization), and Animal's name in italics because it is abstract.

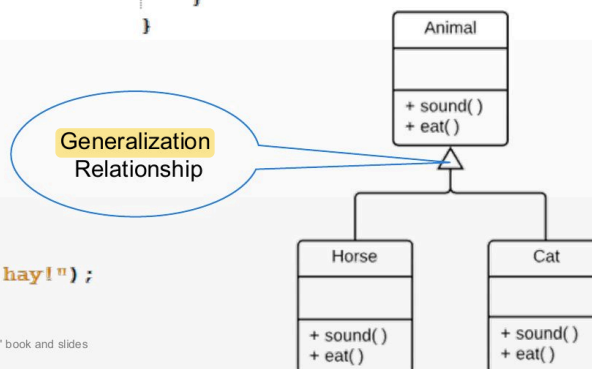
Refresh your memory ... OOP concepts (Inheritance)

```
public class Animal {
    public void sound() {
        System.out.println("Animal is making a sound");
    }
    public void eat() {
        System.out.println("Animal is eating..");
    }
}
```

```
public class Cat extends Animal{
    @Override
    public void sound() {
        System.out.println("Miaw");
    }
    @Override
    public void eat() {
        System.out.println("Cat is eating fish!");
    }
}
```

```
public class Horse extends Animal{
    @Override
    public void sound() {
        System.out.println("Neigh");
    }
    @Override
    public void eat() {
        System.out.println("Horse is eating hay!");
    }
}
```

| Prince Sultan University | Slides prepared from "Software Engineering Design: Theory and Practice" book and slides



12

NOTES

- Here Animal is an ordinary (concrete) class, so every method already has a body:

CODE

WHAT IT DOES

```
public class Animal {
```

A normal class: it can be instantiated.

```
    public void sound(){ ..."Animal is
    making a sound"... }
```

Default sound().

```
    public void eat(){ ..."Animal is
    eating..."... }
```

Default eat().

```
public class Horse extends Animal{
```

Horse is-a Animal.

```
    @Override public void sound(){
    ..."Neigh"... }
```

`@Override` asks the compiler to check that a matching parent method exists, which catches typos.

```
    @Override public void eat(){
    ..."Horse is eating hay!"... }
```

Horse's own eat().

```
public class Cat extends Animal{
    ...sound "Miaw", eat "Cat is eating
    fish!" }
```

Cat overrides both methods the same way.

- Unlike slide 11, nothing forces the subclasses to override: they could inherit Animal's versions unchanged. Overriding is a choice here, not a requirement.
- The UML shows **generalization**: solid lines meeting at one hollow triangle that points at the parent, Animal. Read it as "Horse is an Animal" and "Cat is an Animal".

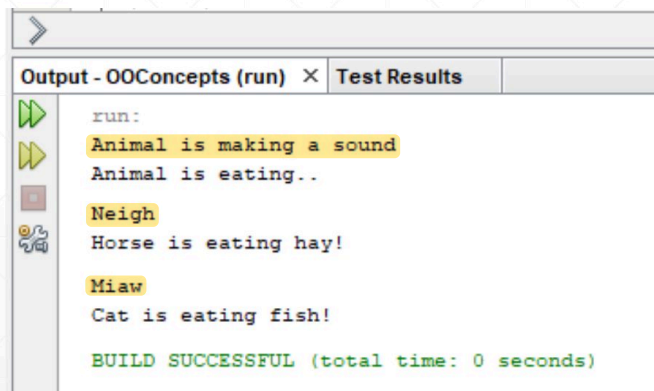
Refresh your memory ... OOP concepts (Inheritance)

```
public class InheritanceDemo {

    public static void main(String[] args) {
        // Animal Object
        Animal a = new Animal();
        a.sound();
        a.eat();

        //Horse Object
        Horse h = new Horse();
        h.sound();
        h.eat();

        //Cat Object
        Cat c = new Cat();
        c.sound();
        c.eat();
    }
}
```



```
run:
Animal is making a sound
Animal is eating..
Neigh
Horse is eating hay!
Miaw
Cat is eating fish!
BUILD SUCCESSFUL (total time: 0 seconds)
```

| Prince Sultan University | Slides prepared from "Software Engineering Design: Theory and Practice" book and slides

13

NOTES

- 1 The demo creates one object of each class and calls both methods:

CODE

```
Animal a = new Animal(); a.sound();
a.eat();
```

WHAT IT DOES

Prints "Animal is making a sound" then "Animal is eating..".

```
Horse h = new Horse(); h.sound();
h.eat();
```

Prints "Neigh" then "Horse is eating hay!".

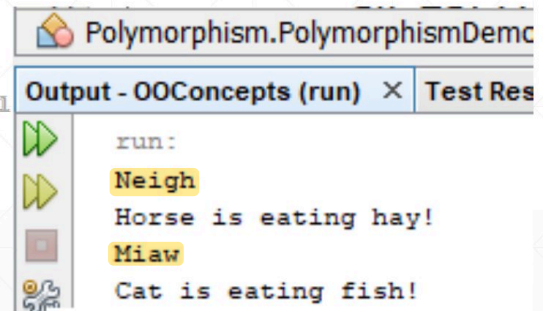
```
Cat c = new Cat(); c.sound(); c.eat();
```

Prints "Miaw" then "Cat is eating fish!".

- 2 The output panel on the right confirms each object runs the version defined in its own class. Every variable here has the same type as its object, so this is plain inheritance. Slide 14 shows what changes when the types differ.

Refresh your memory ... OOP concepts (Polymorphism)

```
public class PolymorphismDemo {  
  
    public static void main(String[] args) {  
  
        //Polymorphism  
        //call the horse constructor to ceate an animal  
        Animal ah = new Horse();  
        ah.sound();  
        ah.eat();  
  
        //Polymorphism  
        //call the cat constructor to ceate an animal  
        Animal ac = new Cat();  
        ac.sound();  
        ac.eat();  
    }  
}
```



| Prince Sultan University | Slides prepared from "Software Engineering Design: Theory and Practice" book and slides

14

NOTES

- 1 The variables are declared as Animal but hold a Horse and a Cat:

CODE

WHAT IT DOES

```
Animal ah = new Horse();
```

Declared type Animal, actual object Horse. This is allowed because a Horse is an Animal.

```
ah.sound(); ah.eat();
```

Prints "Neigh" and "Horse is eating hay!": the HORSE's versions, not Animal's.

```
Animal ac = new Cat();
```

Same idea with a Cat.

```
ac.sound(); ac.eat();
```

Prints "Miaw" and "Cat is eating fish!".

- 2 This is **polymorphism (dynamic binding)**. The declared type (Animal) decides which methods you are allowed to call; the actual object (Horse or Cat) decides which code runs. The choice happens at run time.
- 3 Why it matters for this lecture: OCP (slides 22–31) works because a GameEngine holding a Character reference can call move() and get each character's own movement without knowing its class.

EXAM ANGLE

Prediction

Q1

Animal x = new Cat(); x.sound(); What is printed, and what is this mechanism called?

Answer: "Miaw". It is polymorphism (dynamic binding): the overriding method of the actual object runs, even though the variable's type is Animal.

Design Principles of Internal Component Design

- When designing software at the component-level, several **principles** have to be followed to create designs that **less complex**, **reusable**, **easier to modify**, and **easier to maintain**.
- **Design principles for internal component design (SOLID):**
 - ✓ Single Responsibility Principle (SRP)
 - ✓ Open-Closed Principle (OCP)
 - ✓ Liskov Substitution Principle (LSP)
 - ✓ Interface-Segregation Principle (ISP)
 - ✓ Dependency-Inversion Principle (DIP)

NOTES

- 1 The goal of every principle that follows: designs that are less complex, reusable, easier to modify and easier to maintain.
- 2 SOLID is the acronym for the five principles: **S**ingle Responsibility, **O**pen-Closed, **L**iskov Substitution, **I**nterface Segregation, **D**ependency Inversion. S, O and L are covered in this lecture.

SLIDE 16 / 39 1- Single Responsibility Principle (SRP)

1- Single Responsibility Principle (SRP)

- The **SRP** might be the **least well understood principle**
- Programmers usually assume that it means that **every module should do just one thing**.
- However, **SRP** is described as:
“**A module should be responsible to one, and only one, user or stakeholder.**”
OR “**A class should have only one reason to change.**”

NOTES

- 1 The common misreading: "every module should do just one thing." That is good advice for small functions, but it is **not** what SRP says.
- 2 SRP is about who causes change. A module should answer to **one** actor (a user group, stakeholder or policy). Equivalently, a class has one reason to change. If the accountants and the IT team could each force edits to the same class, it has two responsibilities.

EXAM ANGLE

Prediction

Q1

State the Single Responsibility Principle and explain what "reason to change" means.

Answer: A class should have only one reason to change, i.e. it should be responsible to one actor or stakeholder. A reason to change is a source of change requests: a policy, user group or concern (data format, tax rules, report layout). Two independent sources mean two responsibilities that should be separated.

SLIDE 17 / 39 SRP - example

SRP - example

- The following example is a class that defines a **Person**, this class **should not include email validation** because that is not related with a person behaviour:

```
class Person {
    public name : string;
    public surname : string;
    public email : string;
    constructor(name : string, surname : string, email : string){
        this.surname = surname;
        this.name = name;
        if(this.validateEmail(email)) {
            this.email = email;
        }
        else {
            throw new Error("Invalid email!");
        }
    }
    validateEmail(email : string) {
        var re = /^[a-zA-Z0-9_+&quot;'+@&#39;]+$/;
        return re.test(email);
    }
    greet() {
        alert("Hi!");
    }
}
```

NOTES

1 The code (TypeScript-style), line by line:

CODE	WHAT IT DOES
<code>class Person {</code>	Represents a person.
<code> public name : string; public surname : string; public email : string;</code>	Person data: three text fields.
<code> constructor(name, surname, email){</code>	Runs when a Person is created.
<code> this.surname = surname; this.name = name;</code>	Stores the names.
<code> if(this.validateEmail(email)) { this.email = email; }</code>	Checks the address format before storing it.
<code> else { throw new Error("Invalid email!"); }</code>	Rejects a bad address by throwing an error.
<code> validateEmail(email : string) {</code>	Email-format rule, living inside Person (the boxed part on the slide).
<code> var re = /^[^\w-]+(?:\. [^\w-]+)*)@(\...)/;</code>	A regular expression describing a valid address.
<code> return re.test(email); }</code>	True if the address matches the pattern.
<code> greet() { alert("Hi!"); }</code>	Genuine person behavior.

2 The boxed lines show the violation: Person now has two reasons to change. Its fields change when what a person is changes; `validateEmail` changes when the email rules change (new domains, longer TLDs). You also can't test the email rule without building a Person.

SRP - example

- We can improve the Person class by **removing the responsibility of email validation** from the Person class and creating a new Email class
- Making sure that a class has a **single responsibility (for a single actor)** makes it per default also easier to see what it does and how you can extend/improve it.

```
class Email {
    public email : string;
    constructor(email : string){
        if(this.validateEmail(email)) {
            this.email = email;
        }
        else {
            throw new Error("Invalid email!");
        }
    }
    validateEmail(email : string) {
        var re = /^[^\w-]+(?:\.[^\w-]+)*@((?:[\w-]+\.)*\w[\w-]{0,66})\.([a-z]{2,6})$/;
        return re.test(email);
    }
}
```

```
class Person {
    public name : string;
    public surname : string;
    public email : Email;
    constructor(name : string, surname : string, email : Email){
        this.email = email;
        this.name = name;
        this.surname = surname;
    }
    greet() {
        alert("Hi!");
    }
}
```

| Prince Sultan University | Slides prepared from "Software Engineering Design: Theory and Practice" book and slides

18

NOTES

- 1 The refactoring moves validation into its own class:

CODE	WHAT IT DOES
class Email {	New class that owns everything about an email address.
<pre> constructor(email){ if(this.validateEmail(email)) {...} else { throw ... } }</pre>	An Email object can only exist if the address is valid.
<pre> validateEmail(email){ ... return re.test(email); }</pre>	The same rule, now in its only home.
class Person {	Back to pure person data and behavior.
<pre> public email : Email;</pre>	The field's type is now Email, not string.
<pre> constructor(name, surname, email : Email){ this.email = email; ... }</pre>	Person receives an already-validated Email, so no check is needed.
<pre> greet() { alert("Hi!"); }</pre>	Unchanged.

- 2 Each class now has one reason to change: email-rule changes touch only Email, and person changes touch only Person. A class with one responsibility (one actor) is also easier to read, test and extend.

Class Example	Violates SRP (Has Multiple Reasons to Change)	Follows SRP (Has One Reason to Change)
Employee Class	Stores employee data, and calculates their taxes, and prints their pay stub.	Only stores employee data (name, salary, etc.).
Reason to Change?	Changes if the data structure changes, OR if the tax laws change, OR if the print format changes. (Three reasons!)	Changes ONLY if the employee data structure changes. (One reason!)

does it sound like now for every function i need to make one class??

NOTES

- 1 The violating Employee class stores data, calculates taxes and prints pay stubs. Three different things can force edits: the data structure, the tax laws and the print format.
- 2 The SRP version keeps only the employee data in Employee. Tax calculation and pay-stub printing move to their own classes, so Employee changes only when the data changes.
- 3 The slide's question has a clear answer, given on slide 20: **no**. SRP does not mean one class per function.

➤ SRP Applies to **Responsibilities**, Not **Methods**

- ✓ A class should have **one primary responsibility**, which is usually accomplished by **several related functions (methods)** working together.
- ✓ **Bad Example (Violation):** A single class, ReportGenerator, has methods like calculateTax(), printToPDF(), and sendEmail(). These are three separate *responsibilities* (Calculation, Formatting, Communication).
- ✓ **Good Example (Following SRP):** A class, TaxCalculator, has methods like getGrossSalary(), applyFederalTax(), and applyStateTax(). All three methods work together to fulfill the **single responsibility** of **tax calculation**.

Class	Responsibility	Methods
UserAuthentication	Managing user login state.	login(), logout(), checkSession(), hashPassword()
ProductValidator	Ensuring product data is correct.	validateName(), validatePrice(), validateSKU()

✓ When to Create a New Class:

You should create a new class only when you identify a **new, distinct reason for change**

• If you change method A, do you also need to change method B?

- **Yes:** They likely belong in the same class (e.g., `applyFederalTax()` and `applyStateTax()` both change when tax laws change).
- **No:** They likely belong in separate classes (e.g., `calculateTax()` and `printToPDF()` change for completely different reasons).

20

NOTES

- 1 A responsibility is usually carried out by **several** related methods working together. SRP limits the number of responsibilities, not the number of methods.
- 2 ReportGenerator mixes three responsibilities: calculation (`calculateTax()`), formatting (`printToPDF()`) and communication (`sendEmail()`). Each changes for a different reason.
- 3 TaxCalculator has three methods but one responsibility, tax calculation. The table adds two more good examples: UserAuthentication (login state, four methods) and ProductValidator (product data correctness, three methods).
- 4 The test for splitting: if you change method A, would you also have to change method B? If yes, they belong together (both tax methods change with tax law). If no, they belong in separate classes (`calculateTax()` and `printToPDF()`).

In Class Exercises

- **Design Exercise: Employee Payroll System, Principles to Apply, Single Responsibility Principle (SRP):** Each class should have only one reason to change and
 - We need to design a class diagram for a simple **Employee Payroll System**.
 - The initial requirements are:
 - ✓ The system needs to **calculate the salary** for an employee.
 - ✓ The system needs to **save the employee's data** to a database.
 - ✓ The system needs to **generate a report** (e.g., a simple printout) of the employee's details.
 - **Questions:**
 - 1- **Draw a class diagram that violates SRP.**
 - 2- **Redraw to achieve SRP.**

NOTES

- 1 Three requirements means three potential reasons to change: pay rules, storage technology and report format.
- 2 Question 1, a design that violates SRP (one class does everything):


```
Employee
- id, name, baseSalary
+ calculateSalary(): double
+ saveToDatabase(): void
+ generateReport(): String
```
- 3 Question 2, the SRP redesign. Employee keeps the data; each job gets its own class, and each one depends on (uses) Employee:

```
Employee
- id, name, baseSalary          (data only)

SalaryCalculator
+ calculateSalary(e: Employee): double

EmployeeRepository
+ save(e: Employee): void

EmployeeReportGenerator
+ generate(e: Employee): String

all three - - -> Employee    (dependency)
```

Past exam · Final 252 · Q5-1 · 2.5 marks

Q1

NextGen Library: a junior developer put fee calculation, book data and report formatting into one BookManager class. Which detailed-design principle is violated? Draw a class diagram that fixes it.

Answer: The Single Responsibility Principle: BookManager has three reasons to change (book data, loan and fee rules, report format). Fix: Book (title, ISBN, author), LoanCalculator (due date and late fee) and MonthlyReportGenerator (text report), each with one responsibility; LoanCalculator and the report class use Book.

SLIDE 22 / 39 2- The Open-Closed Principle (OCP)

2- The Open-Closed Principle (OCP)

- The **Open-Closed principle** (OCP) is an essential principle for **creating reusable** and **modifiable** systems that evolve gracefully with time.
- The OCP states that “**software designs should be open to extension but closed for modification.**”
 - ✓ The main idea behind the OCP is that code that works should remain **untouched** and that new additions should be extensions of the original work.
- That sounds contradictory, how can that be?
 - ✓ Being close to modifications does not mean that designs cannot be modified; it means that **modifications should be done by adding new code, and incorporating this new code in the system in ways that does not require old code to be changed!**

NOTES

- 1 OCP is what lets a system keep growing for years without every new feature breaking old ones.
- 2 “Open to extension” means you can add new behavior. “Closed for modification” means code that already works and is tested stays as it is.
- 3 There is no contradiction: the system still changes, but by **adding** classes (a new subclass, a new implementation) that plug into existing abstractions. No working code has to be edited for that.

Prediction

Q1

Explain how a design can be both "open for extension" and "closed for modification".

Answer: New behavior is added as new code (new classes implementing an existing interface or extending an abstract class) that the system uses through that abstraction, so existing, tested classes are never edited.

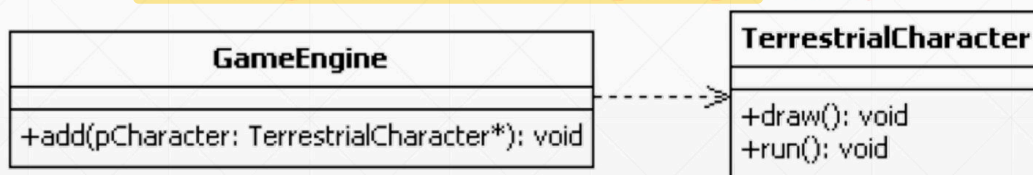
SLIDE 23 / 39 The Open-Close Principle

The Open-Close Principle

➤ A gaming system Example

Consider a fictional **gaming system** that includes **several types of terrestrial characters**, ones that can roam freely over land. *It is anticipated that new characters will be added in the future.*

This design violates the OCP principle ... why?



| Prince Sultan University | Slides prepared from "Software Engineering Design: Theory and Practice" book and slides

23

NOTES

- 1 The scenario matters: several kinds of characters exist, and **more will be added**. A change is expected, so the design must be ready for it.
- 2 Reading the diagram:
 - GameEngine**
Has one operation, `add(pCharacter: TerrestrialCharacter*)`. The `*` is a C++ pointer.
 - Dashed arrow**
A dependency: GameEngine uses TerrestrialCharacter directly.
 - TerrestrialCharacter**
A concrete class with `draw()` and `run()`.
- 3 Why it violates OCP: GameEngine names a **concrete** class. Every new kind of character (one that flies, one that swims) forces a change to GameEngine, the code that already works.

The Open-Close Principle

Note:

This is really not the code for a gaming system! The code is for illustration purpose.

```
// The terrestrial character.
class TerrestrialCharacter {

public:
    // Draw the character on the screen.
    virtual void draw() { /*Code to draw the terrestrial character.*/

    // Make the character run!
    virtual void run() { /* Code to make the character run.*/
};
```

```
// The game engine responsible for managing the game.
class GameEngine {

public:
    // Add the character to the screen.
    void add(TerrestrialCharacter* pCharacter) {

        // Display the character.
        pCharacter->draw();

        // Make the character move!
        pCharacter->run();
    }
};
```

What can you tell me about the add(...) function?

What happens if we add a new requirement to support other types of characters, e.g., an AerialCharacter that can fly?

Yes, that is right, we would have to change the code inside the add(...) method. This violates the OCP ! Let's see an improved version in the next slide...

NOTES

- 1 The C++ code, line by line:

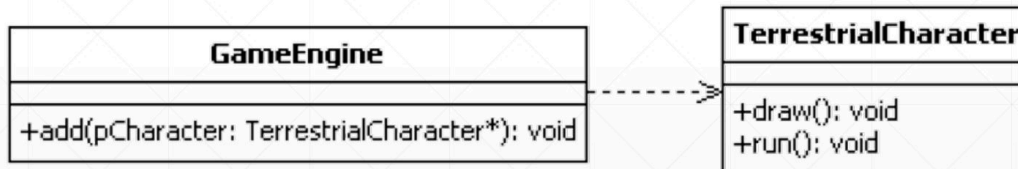
CODE	WHAT IT DOES
class TerrestrialCharacter {	A character that moves over land.
virtual void draw() { ... }	<code>virtual</code> = subclasses may override it, and the call is resolved at run time.
virtual void run() { ... }	Land movement.
class GameEngine {	Manages the game.
void add(TerrestrialCharacter* pCharacter) {	Accepts only terrestrial characters.
pCharacter->draw();	-> calls a method through a pointer: show the character.
pCharacter->run();	Make it move by running.

- 2 `add(...)` is hard-wired to one concrete type and one way of moving (`run()`).
- 3 An AerialCharacter that flies doesn't fit: it isn't a TerrestrialCharacter and has no `run()`. Supporting it means editing `add(...)`, which is working code. That edit is the OCP violation.

The Open-Close Principle (OCP)

This design violates the OCP principle ... why?

Do you have suggestions to fix it to allow adding new types of Characters without violating the OCP?

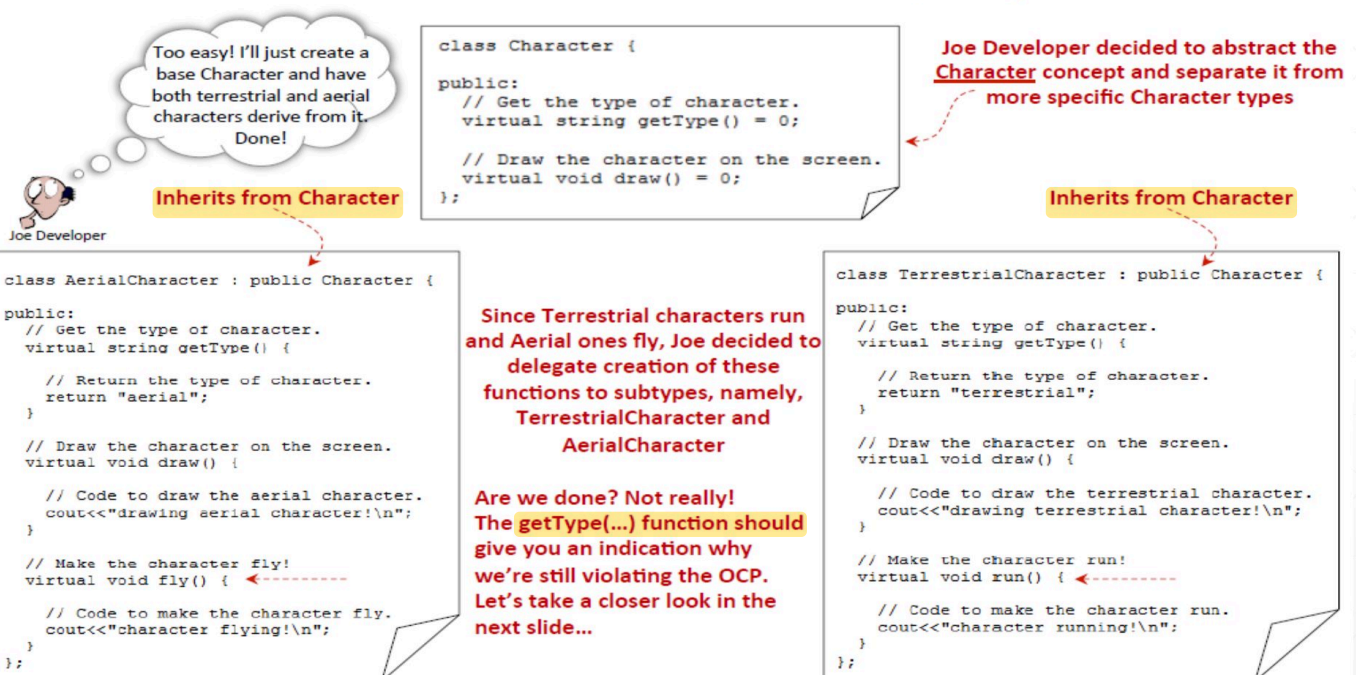


NOTES

- 1 The fix students should propose: introduce an abstraction (an interface or abstract class) named **Character** with the operations every character has, and make GameEngine depend on Character instead of on a concrete class. Slides 26–29 show how a first attempt still goes wrong, and then the correct version.

SLIDE 26 / 39 The Open-Closed Principle (OCP): first attempt

THE OPEN-CLOSED PRINCIPLE (OCP)



Note: Character is really an interface, so instead of "Inherits from Character" it (more precisely) realizes the Character interface.

NOTES

1 Joe's first attempt (C++):

CODE	WHAT IT DOES
<pre>class Character { virtual string getType() = 0; virtual void draw() = 0; };</pre>	= 0 makes a method pure virtual (abstract). A class with only pure virtual methods acts as an interface.
<pre>class AerialCharacter : public Character {</pre>	: public Character = inherits from (realizes) Character.
<pre> getType() { return "aerial"; } draw() {...} fly() {...}</pre>	Reports its type, draws itself, and has its own fly() .
<pre>class TerrestrialCharacter : public Character {</pre>	Same shape:
<pre> getType() { return "terrestrial"; } draw() {...} run() {...}</pre>	its own run() .

2 Joe abstracted the drawing, but movement was left to each subtype under **different names** (fly , run). Character itself has no movement operation.

3 The getType() method is the warning sign: a caller that has to ask "what type are you?" is about to branch on the answer. Slide 27 shows that branch.

SLIDE 27 / 39 The Open-Closed Principle (OCP): GameEngine with type checks

THE OPEN-CLOSED PRINCIPLE (OCP)

Design Principle:
Encapsulate Variation

Notice how the GameEngine client needs to know the type of Character before it can activate it. This is a side-effect of a violation of the OCP

 Yikes!

```
class GameEngine {
public:
    // Add a character to the game.
    void add( Character* pCharacter ) {
        // Draw the character on the screen.
        pCharacter->draw();

        // If aerial, make it fly, otherwise, make it run.
        if( pCharacter->getType().compare("aerial") == 0 ) {
            // Downcast the pointer to an aerial character.
            AerialCharacter* pAerial = dynamic_cast<AerialCharacter*>(pCharacter);

            // Assume a valid pointer and make the character fly!
            pAerial->fly();
        }
        else {
            // Downcast the pointer to a terrestrial character.
            TerrestrialCharacter* pTerrestrial =
                dynamic_cast<TerrestrialCharacter*>(pCharacter);

            // Make the character run!
            pTerrestrial->run();
        } // end if statement.
    } // end add function.
};
```

This code will **always vary**, depending on the **characters in the game!**

Sample test driver code

```
int _tmain(int argc, _TCHAR* argv[])
{
    // create the mad rabbit character.
    TerrestrialCharacter madRabbit;
    // create the killer bee character.
    AerialCharacter killerBee;

    // create the game engine.
    GameEngine engine;

    // add characters to the game.
    engine.add(&madRabbit);
    engine.add(&killerBee);
    system("pause");

    return 0;
}
```

Sample output

```
drawing terrestrial character!
character running!
drawing aerial character!
character flying!
Press any key to continue . . .
```

It works! We're done!
Not really, we've improved the design, but are we OCP-Compliant?

The Character design still requires clients to know too much about Characters. What would happen if we now need to support an **Aquatic Character**?

Let's see in the next slide how to make this design OCP-Compliant...

NOTES

1 GameEngine.add with Joe's design:

CODE	WHAT IT DOES
<code>void add(Character* pCharacter) {</code>	Now takes any Character, which is better.
<code> pCharacter->draw();</code>	Drawing works for every character.
<code> if(pCharacter->getType().compare("aerial") == 0) {</code>	Asks for the type string.
<code> AerialCharacter* pAerial = dynamic_cast<AerialCharacter*>(pCharacter);</code>	<code>dynamic_cast</code> = a checked downcast to the concrete type.
<code> pAerial->fly(); }</code>	Only then can it call fly().
<code> else { TerrestrialCharacter* pTerrestrial = dynamic_cast<...>(pCharacter);</code>	Otherwise assume terrestrial...
<code> pTerrestrial->run(); }</code>	...and run.
<code>test driver: TerrestrialCharacter madRabbit; AerialCharacter killerBee; engine.add(&madRabbit); engine.add(&killerBee);</code>	Prints: drawing terrestrial / running / drawing aerial / flying.

2 **Encapsulate variation:** the part that changes (how each character moves) should be hidden behind one stable operation. Here it leaks into GameEngine as an if/else that "will always vary" with the set of characters.

3 It runs correctly, but it still violates OCP: adding an AquaticCharacter means editing `add()` again with another branch and another cast.

THE OPEN-CLOSED PRINCIPLE (OCP)

```
class Character {
public:
    // Draw the character on the screen.
    virtual void draw() = 0;

    // Make the character move.
    virtual void move() = 0;
};
```

Encapsulate the movement behavior, so that `move(...)` works for all characters in the game!

```
// The aerial character.
class AerialCharacter : public Character {
public:
    // Draw the character on the screen.
    virtual void draw() { /* Code to draw the aerial character. */ }

    // Make the character fly.
    virtual void move() { /* Code to make the character fly! */ }
};
```

Per the interface contract, these must provide the implementation for both draw and fly services

```
// The terrestrial character.
class TerrestrialCharacter : public Character {
public:
    // Draw the character on the screen.
    virtual void draw() { /* Code to draw the terrestrial character. */ }

    // Make the character run.
    virtual void move() { /* Code to make the character run! */ }
};
```

In the next slide, let's see how the code for the GameEngine class looks now based on this new design...

NOTES

1 The fix: put one movement operation, `move()`, into Character itself (`virtual void move() = 0`). "How it moves" becomes each subtype's private business.

2 Each subtype implements the full contract:

AerialCharacter

`draw()` draws it, `move()` makes it fly.

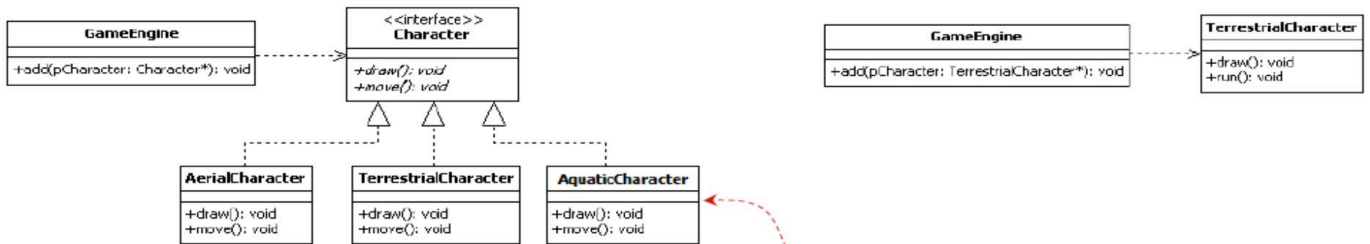
TerrestrialCharacter (sic)

`draw()` draws it, `move()` makes it run.

THE OPEN-CLOSED PRINCIPLE (OCP)

New redesign! Adheres to OCP!

Old design! Violates OCP!



```
// The game engine responsible for managing the game.
class GameEngine {
public:
    // Add the character to the screen.
    void add(Character* pCharacter) {
        // Display the character.
        pCharacter->draw();

        // Activate the character... make it move!
        pCharacter->move();
    } // end add function.
};
```

New Aquatic Character added by extension and not by modifying existing working code!

With this design, GameEngine can draw and activate current and future Characters in the game without modification!

NOTES

- Reading the new diagram (left) against the old one (right):
 - GameEngine - - → «interface» Character**
GameEngine depends only on the interface. The italic `+draw()` and `+move()` are abstract operations.
 - Aerial / Terrestrial / AquaticCharacter**
Each realizes Character (dashed line, hollow triangle) and implements both operations.
 - Old design**
GameEngine depended on the concrete TerrestrialCharacter.

- AquaticCharacter is added by **writing one new class**. GameEngine, Character and the existing characters are not touched. That is OCP.

- 3 add() now just calls draw() and move(); polymorphism (slide 14) runs the right movement. The same design in Java:

```
interface Character {
    void draw();
    void move();
}
class AquaticCharacter implements Character {
    public void draw() { /* draw a fish */ }
    public void move() { /* swim */ }
}
class GameEngine {           // never changes
    void add(Character c) {
        c.draw();
        c.move();
    }
}
```

SLIDE 30 / 39 The Open-Closed Principle (OCP)

The Open-Closed Principle (OCP)

One final note about the OCP:

No design will be 100% closed for modification. At some point, some code has to be readily-available for tweaking in any software system. The idea of the OCP is to locate the areas of the software that are **likely to vary** and the variations can be encapsulated and implemented through **polymorphism**

NOTES

- 1 OCP is a direction, not an absolute. Something always has to be edited: a factory, a configuration or a list of characters.
- 2 The skill is to predict **where** change is likely (new character types, new salary types, new payment methods), hide that variation behind an abstraction, and let polymorphism pick the implementation.

Exercise

- **Open-Closed Principle (OCP):** Software entities (classes, modules, functions, etc.) should be **open for extension but closed for modification**.
- Going back to the pervious exercise on SRP, Specifically, the salary calculation logic must be designed so that adding new types of Salaries (e.g., **Hourly** or **weekly** salary..etc) doesn't require changing the main salary calculation class.
- Open the same design you did and **apply OCP** in there .

NOTES

- 1 The same rule restated for any software entity: a class, module or function.
- 2 Model answer: make the varying part (how a salary is computed) an interface, so a new salary type is a new class.

```
«interface» SalaryPolicy
+ calculate(e: Employee): double

MonthlySalary, HourlySalary, WeeklySalary
- - -|> SalaryPolicy          (realization)

SalaryCalculator                (never edited)
+ calculateSalary(e: Employee,
                  p: SalaryPolicy): double

New CommissionSalary = one new class
that implements SalaryPolicy.
```

Past exam · Final 252 · Q5-2 · 2.5 marks

Q1

NextGen Library (physical books now, E-books soon): by applying the Open/Closed Principle for E-books, draw the class diagram.

Answer: Introduce an abstract class or interface (e.g. `LibraryItem` with title, ISBN, author and an abstract operation such as `calculateLateFee()`). `PhysicalBook` and `EBook` extend or implement it. The loan and report classes depend only on `LibraryItem`, so adding `EBook` (or audiobooks later) needs no change to existing classes.

SLIDE 32 / 39 **3- The Liskov Substitution Principle (LSP)**

3- The Liskov Substitution Principle (LSP)

- LSP demands that “Any class derived from a base class must honor any **implied contract** between the base class and the components that use it.”
- In other words: “Objects should be **replaceable with instances of their subtypes** without altering the correctness of that program” (The **"Treat Alike" Rule**)
- The LSP requires:
 - ✓ **Signatures** between base and derived classes to be maintained
 - ✓ **Subtype specification** supports reasoning based on the super type specification

| Prince Sultan University | Slides prepared from "Software Engineering Design: Theory and Practice" book and slides

32

NOTES

- 1 The “implied contract” is everything callers rely on: the method names and types, but also the **behavior** (what the method promises to do and what it requires before it is called).
- 2 If code works with a parent object, it must keep working when you hand it a child object instead. The slide calls this the “treat alike” rule.
- 3 Two requirements: the method signatures stay compatible, and anything you could conclude from the parent’s specification must still hold for the subtype.

Prediction

Q1

State the Liskov Substitution Principle in one sentence.

Answer: Objects of a superclass must be replaceable by objects of any subclass without altering the correctness of the program: subtypes must honor the base type's contract.

SLIDE 33 / 39 The Liskov Substitution Principle (LSP)

The Liskov Substitution Principle (LSP)

- To adhere to the LSP, designs must conform to the following rules:
 - ✓ **The Signature Rule:** ensures that if a program is type-correct based on the super type specification, it is also type-correct with respect to the subtype specification.
 - ✓ **The Methods Rule:** ensures that reasoning about calls of super type methods is valid even though the calls actually go to code that implements a subtype.
- Imagine you have a family of objects. The **Liskov Substitution Principle (LSP)** is just a fancy way of saying:
- **If a class is an extension of another class, you must be able to use the new class in place of the original class without anything breaking.**

| Prince Sultan University | Slides prepared from "Software Engineering Design: Theory and Practice" book and slides

33

NOTES

- 1 **Signature rule:** if a program type-checks against the parent's methods, it type-checks with the child too (same methods with compatible parameters and return types). The compiler mostly enforces this.
- 2 **Methods rule:** reasoning about a call stays valid even though the child's code runs. If `move()` promises "moves over land", every override must still move over land. The compiler can't check this, so the designer must.
- 3 The plain-language version: a child class must work anywhere the parent works, without anything breaking.

Prediction

Q1

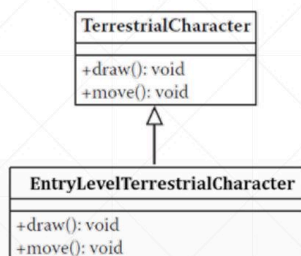
Which LSP rule does the compiler NOT check for you, and why?

Answer: The methods rule (behavior). It concerns what methods do: pre- and postconditions and side effects. Types alone can't express that, so a subclass can compile and still break callers.

SLIDE 34 / 39 The Liskov Substitution Principle (LSP) - example

The Liskov Substitution Principle (LSP) - example

Does this example apply LSP?



In some cases, it can be seen that adhering to the OCP alone does not guarantee correct designs or designs that lead to reusability throughout the system.

C++ Implementation for EntryLevelTerrestrialCharacter

```

class TerrestrialCharacter : public Character {
    // Pre-Condition: Character is drawn on screen.
    // Post-Condition: Character roams around randomly over land, either
    // walking or running.
    virtual void move() {
        //code to make the character walk or run.
    }
    // code here to implement all other interfaces.
};
  
```

```

class EntryLevelTerrestrialCharacter : public TerrestrialCharacter {
public:
    // Override move to make the character fly!
    virtual void move() {
        // code to make the character fly.
    }
    // Code here to implement all other interfaces.
};
  
```

NOTES

1 The code and its contract:

CODE

WHAT IT DOES

```
class TerrestrialCharacter : public
Character {
```

The parent.

```
    // Pre-Condition: Character is drawn
on screen.
```

What must be true before calling `move()`.

```
    // Post-Condition: Character roams
around randomly over land...
```

What `move()` promises: it stays on land, walking or running.

```
    virtual void move() { //code to make
the character walk or run }
```

Honors the promise.

```
class EntryLevelTerrestrialCharacter :
public TerrestrialCharacter {
```

A "kind of" terrestrial character.

```
    virtual void move() { // code to
make the character fly. }
```

Overrides `move()` to FLY.

- 2 **Answer: no, it violates LSP.** It compiles (signature rule OK) but breaks the postcondition “roams over land”. Code that relies on terrestrial characters staying on the ground, such as collision with water or bridges, misbehaves when given this subclass.
- 3 The lesson in red: OCP alone isn’t enough. The class was added by extension (OCP satisfied), yet the design is wrong because the subtype isn’t substitutable. A similar trap is a subclass whose `move()` throws an exception (“Ghosts don’t move!”): that also breaks the contract.

EXAM ANGLE

Prediction

Q1

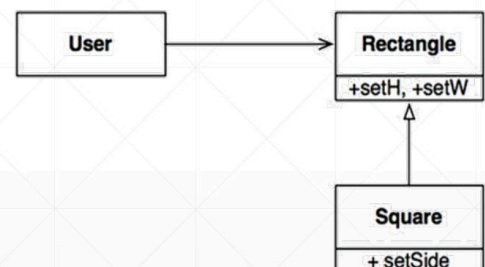
A subclass compiles and overrides every method, but one override throws `UnsupportedOperationException`. Does it satisfy LSP?

Answer: No. It satisfies the signature rule but violates the methods rule: callers of the base type expect the operation to work. The subclass can’t replace its parent without breaking the program.

SLIDE 35 / 39 The Liskov Substitution Principle (LSP) – example 2

The Liskov Substitution Principle (LSP) – example 2

- An example of a violation of the LSP is the famed **square/rectangle problem**
- In this example, Square is not a proper subtype of Rectangle because the height and width of the Rectangle are **independently mutable**;
- in contrast, the height and width of the Square **must** change together.
- Since the User believes it is communicating with a Rectangle, it could easily get **confused**.



NOTES

- 1 The classic LSP violation. Mathematically every square is a rectangle, but in software Square should not inherit from Rectangle.
- 2 Rectangle’s implied contract: `setH` and `setW` change only their own dimension. A user that sets width 5 and height 2 expects an area of 10.
- 3 A Square has to keep its sides equal, so its setters change both. After `setW(5); setH(2)` the area is 4, not 10.

4 Diagram and fix:

User → Rectangle

Solid arrow: User is associated with and uses Rectangle (`+setH` , `+setW`).

Square → Rectangle (hollow triangle)

Generalization: Square inherits from Rectangle and adds `+setSide` .

Fix

Don't inherit. Make Square and Rectangle siblings under a Shape abstraction that offers only what both can honor, e.g. `area()` .

EXAM ANGLE

Prediction

Q1

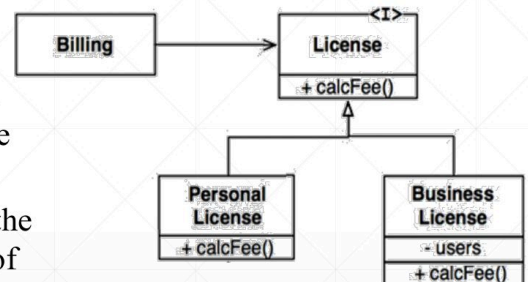
Why does Square extends Rectangle violate LSP? Use a short example.

Answer: Rectangle lets width and height change independently. Code like `r.setW(5); r.setH(2); assert r.area()==10` works for a Rectangle, but a Square forces both sides equal, so the area is 4 and the assertion fails. A Square can't substitute for a Rectangle.

SLIDE 36 / 39 The Liskov Substitution Principle (LSP) – example 3

The Liskov Substitution Principle (LSP) – example 3

- Imagine that we have a class named License. This class has a method named `calcFee()`, which is called by the Billing application.
- There are two “subtypes” of License: PersonalLicense, and BusinessLicense. They use different algorithms to calculate the license fee.
- This design conforms to the LSP because the behaviour of the Billing application does not depend, in any way, on which of the two subtypes it uses.
- Both of the subtypes are substitutable for the License type.



NOTES

1 Reading the diagram:

Billing → License

Billing uses License and calls `calcFee()`.

<I> License

An interface with one operation, `+ calcFee()`.

PersonalLicense, BusinessLicense

Implement License. BusinessLicense also keeps `- users` (a private field) because its fee depends on the number of users.

2 Each subtype computes the fee differently (a flat personal fee; a fee per business user), but both honor the contract "return this license's fee".

3 Billing's behavior never depends on which subtype it receives, so both are substitutable: LSP holds. It is also OCP-compliant, because a new license type is a new class. In Java:

```
interface License { double calcFee(); }

class PersonalLicense implements License {
    public double calcFee() { return 100.0; }
}

class BusinessLicense implements License {
    private int users;
    public double calcFee() {
        return users * 50.0;
    }
}

class Billing {
    void printBill(License l) {
        System.out.println(l.calcFee());
    }
}
```

- The **Liskov Substitution Principle** is an **important sanity check**. It makes sure that your inheritance structure isn't just about reusing code, but about maintaining **expected behavior**

NOTES

- 1 Use LSP to sanity-check every inheritance arrow you draw: ask whether the child can stand in for the parent everywhere.
- 2 Inheritance is about keeping **expected behavior**, not just reusing code. If the honest answer is "only sometimes", use composition or a different abstraction instead.

SLIDE 38 / 39 Exercise

Exercise

- Let's use the Fruit analogy, draw and tell if LSP is violated or not.
- **Base Class (Parent):** Fruit
- ✓ It has a method: Peel()
 - ✓ It has a method: MakeSmoothie()
- **Derived Class (Child 1):** Orange
- ✓ Peel()
 - ✓ MakeSmoothie()
 - ✓ LSP is Applied ??
- ✓ **Derived Class (Child 2):** Coconut
- ✓ Peel()...
 - ✓ MakeSmoothie()...
 - ✓ LSP is Applied??

NOTES

- 1 Fruit promises two operations every fruit can do: `Peel()` and `MakeSmoothie()`.
 - 2 Orange: peeling and blending both work exactly as Fruit promises. **LSP holds.**
 - 3 Coconut: you don't peel a coconut like other fruit; you crack it open. If `Coconut.Peel()` throws an error or does something callers don't expect, a program that peels any Fruit breaks. **LSP is violated.**
- Model drawing and fix:

Violation

```
Fruit (+Peel(), +MakeSmoothie())  
  ^ Orange    both work  
  ^ Coconut   Peel() fails
```

Fix

```
«interface» Peelable (+Peel())  
«interface» Blendable (+MakeSmoothie())  
Orange implements Peelable, Blendable  
Coconut implements Blendable  
and adds its own +Crack()
```

EXAM ANGLE

Prediction

Q1

Does Coconut violate LSP if Fruit declares `Peel()`? How would you fix the design?

Answer: Yes, if Coconut can't honor `Peel()` (it throws or behaves differently), code written for Fruit breaks. Fix: don't promise `Peel()` for every fruit. Split the capabilities into separate interfaces (`Peelable`, `Blendable`) and have each fruit implement only what it can do. This also previews the Interface Segregation Principle.

Lecture Outlines

- **Overview of Component Design**
- **Designing Internal **Structure** of Components (OO Strategy)**
- **Design Principles for Internal Component Design (SOLID)**
 - ✓ Single Responsibility Principle
 - ✓ Open-Closed Principle
 - ✓ Liskov Substitution Principle
 - ✓ Interface-Segregation Principle (Next lecture)
 - ✓ Dependency-Inversion Principle (next lecture)
- **Designing Internal **Behavior** of Components (next lecture)**

NOTES

- 1 Covered so far: component design, the OO refresher, and S, O and L. The **Interface Segregation** and **Dependency Inversion** principles come in the next lecture.
- 2 Also next lecture: designing the **internal behavior** of components.