

Software Design and Architecture

[Principles of Detailed Design] – Chapter 05, L02

Lecture Outlines

- **Overview of Component Design**
- **Designing Internal Structure of Components (OO Strategy)**
- **Design Principles for Internal Component Design (SOLID)**
  - **S**ingle Responsibility Principle
  - **O**pen-Closed Principle
  - **L**iskov Substitution Principle
  - **I**nterface-Segregation Principle
  - **D**ependency-Inversion Principle
- **Designing Internal Behavior of Components**

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.

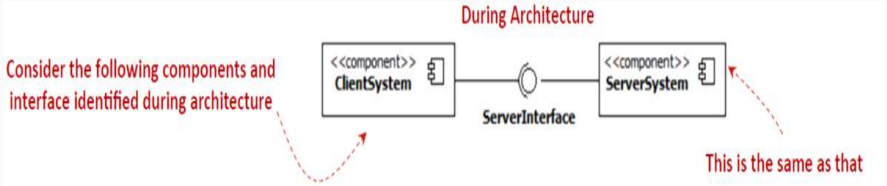
➢ In object-oriented systems, the internal structure of components is typically modeled using UML through one or more class diagrams

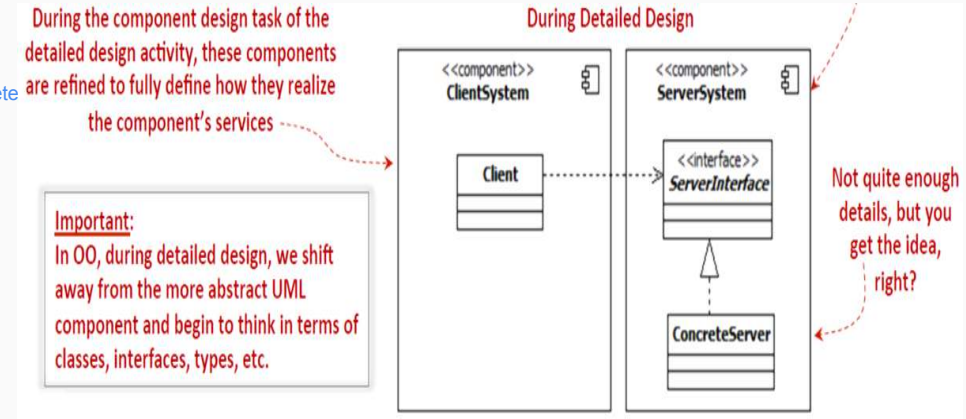
Overview of Component Design

- **Extra:**
  Architecture Phase: Defines high-level components and interfaces (e.g., ClientSystem, ServerSystem, ServerInterface).

  Detailed Design Phase: Refines these into internal details—ClientSystem becomes classes, ServerSystem implements ServerInterface as ConcreteServer.

  Shift in Focus: Moves from abstract components to concrete classes and types, guiding developers in actual implementation.





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
- This allows us to evaluate alternative solutions before construction begins.

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**

Refresh your memory … OOP concepts (Inheritance)

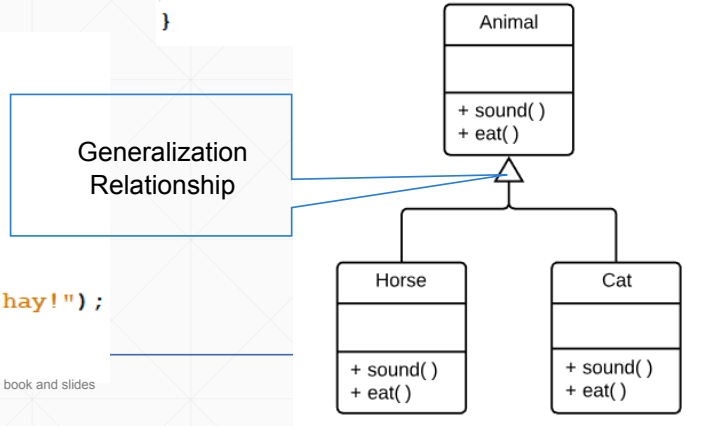
```java
public class Animal {
    public void sound(){
        System.out.println("Animal is making a sound");
    }
    public void eat(){
        System.out.println("Animal is eating..");
    }
}
```

```java
public class Cat extends Animal{
    @Override
    public void sound(){
        System.out.println("Miaw");
    }

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

```java
public class Horse extends Animal{
    @Override
    public void sound(){
        System.out.println("Neigh");
    }

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



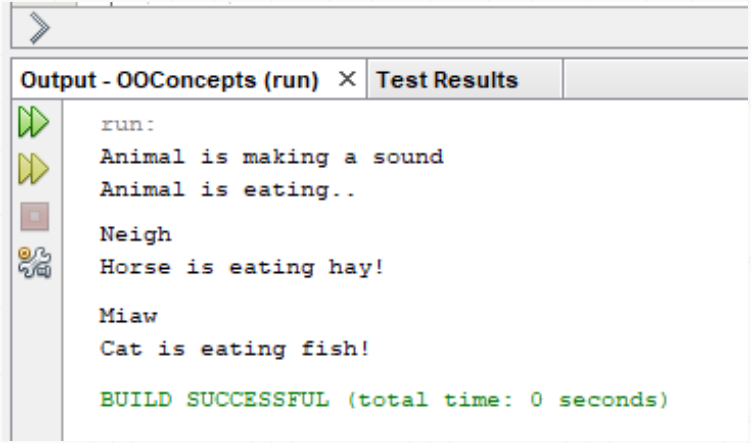
Refresh your memory … OOP concepts (Inheritance)

```
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();
    }
}
```



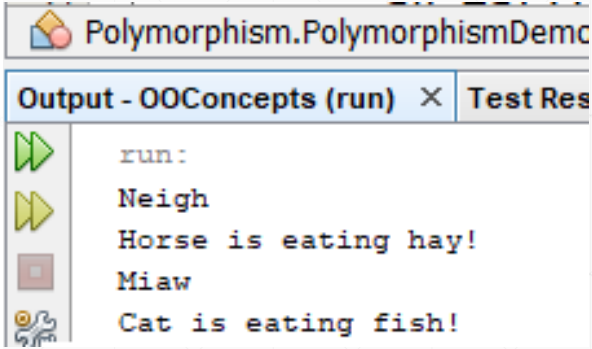
Refresh your memory … OOP concepts (Polymorphism)

```
public class PolymorphismDemo {

    public static void main(String[] args) {

        //Ploymorphism
        //call the horse constructor to ceate an animal
        Animal ah = new Horse();
        ah.sound();
        ah.eat();

        //Ploymorphism
        //call the cat constructor to ceate an animcal
        Animal ac = new Cat();
        ac.sound();
        ac.eat();
    }
}
```



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.

Extra: **1) What is an Abstract Class?**

An **abstract class** in Java is a class that is used as a **base class** for other classes.

You **cannot create an object directly** from an abstract class, but you **can inherit** from it to make subclasses.

An abstract class can contain **two types of methods**:

A) Concrete Method (Normal Method)

- Has a body { }
- Already implemented
- Subclasses can use it directly or override it.

B) Abstract Method

- Has **no body**
- Must be implemented (overridden) inside subclasses.

Also, a subclass can have its own **extra methods** (normal methods) in addition to overriding.

```java
abstract class Animal {
    // Attributes (different types)
    protected String name;                  // normal attribute
    protected int age;                      // normal attribute
    public static int animalCount = 0;      // static attribute (shared)
    public final String category = "Living Creature"; // final constant

    // Constructor (allowed in abstract class)
    public Animal(String name, int age) {
        this.name = name;
        this.age = age;
        animalCount++;
    }

    // ✅ Concrete method (implemented)
    public void showInfo() {
        System.out.println("Name: " + name + ", Age: " + age);
    }

    // ✅ Abstract method (no body → must be implemented in subclass)
    public abstract void makeSound();
}
```

- **Attributes allowed in an abstract class**

An abstract class can contain attributes like:

- `private`, `protected`, `public`
- `final` (constant value)
- `static` (shared between all objects)
- normal instance variables

✅ Subclass: `Dog`

java

```java
class Dog extends Animal {

    public Dog(String name, int age) {
        super(name, age); // call constructor of abstract class
    }

    // ✅ Implementing the abstract method
    @Override
    public void makeSound() {
        System.out.println(name + " says: Woof Woof!");
    }

    // ✅ Subclass own method (extra method)
    public void fetch() {
        System.out.println(name + " is fetching the ball!");
    }
}
```

✅ Main Class (Creating object from subclass)

java

```java
public class Main {
    public static void main(String[] args) {

        // ❌ Not allowed:
        // Animal a = new Animal("Test", 2);

        // ✅ Allowed: object from subclass
        Dog d1 = new Dog("Buddy", 3);

        d1.showInfo();     // concrete method from abstract class
        d1.makeSound();    // overridden abstract method
        d1.fetch();        // subclass extra method

        System.out.println("Category: " + d1.category); // final attribute
        System.out.println("Total animals created: " + Animal.animalCount);
    }
}
```

✅ Expected Output

yaml

```yaml
Name: Buddy, Age: 3
Buddy says: Woof Woof!
Buddy is fetching the ball!
Category: Living Creature
Total animals created: 1
```

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{

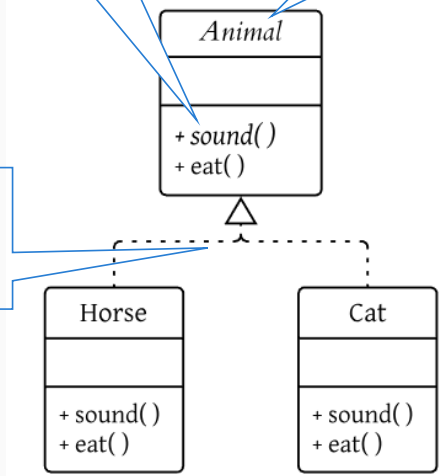
    public void sound(){
        System.out.println("Miaw");
    }

    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

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):**

- **S**ingle Responsibility Principle (SRP)
- **O**pen-Closed Principle (OCP)
- **L**iskov Substitution Principle (LSP)
- **I**nterface-Segregation Principle (ISP)
- **D**ependency-Inversion Principle (DIP)

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

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

Extra:

Why we need Interfaces

1. To **define a contract/blueprint**: "Any class that implements me must provide these methods."
2. To allow **multiple inheritance of type** (a class can implement many interfaces, but only extend one class).
3. To achieve **loose coupling** → the code depends on interfaces (contracts) not concrete implementations, making systems flexible and easier to extend.

Abstract Class vs Interface

Feature	Abstract Class	Interface
Methods	Can have both abstract (no body) and concrete (with body) methods	Originally only abstract methods (Java 8+ allows `default` & `static` methods with body)
Variables	Can have instance variables	Only `public static final` (constants)
Inheritance	A class can extend **only one** abstract class	A class can implement **multiple** interfaces
Constructor	Can have a constructor	Cannot have constructors
Use Case	When classes share **common state + behavior** (fields + reusable methods)	When you just want to define a **contract** without worrying about implementation
Coupling	Tighter → carries some implementation	Looser → only defines method signatures

Refresh your memory … OOP concepts (Interface)

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

}
```

```
public class Cat implements AnimalInterface{

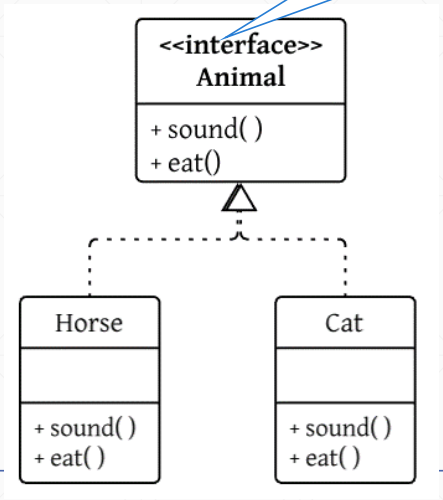
    public void sound(){
        System.out.println("Miaw");
    }

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

}
```

Use the **implements** reserved word in Java

Use the <<interface>> word in UML



Single Responsibility Principle (SRP)

➢ The **SRP** might be the least well understood principle (that's likely because it has a particularly inappropriate name).

➢ Programmers usually assume that it means that every module should do just one thing.

➢ Extra:The **Single Responsibility Principle (SRP)** is one of the SOLID principles in object-oriented design, stating that a class should have only **one reason to change**, meaning it should only have one job or responsibility. This keeps the design more maintainable, understandable, and flexible to changes.

"A module should be responsible to one, and only one, user or stakeholder."

```java
class Student {

    public void storeData() {

        // save student data

    }

    public void calculateGrades() {

        // calculate grades

    }

    public void printReport() {

        // print student report

    }
}
```

Single Responsibility Principle (SRP) — Simple Meaning
A **class** should do **only one job**.
If a class is doing **more than one job**, then every time something changes in one job, the whole class has to change → this makes the design messy.

🔷 **Real-Life Analogy**

Imagine a **restaurant** 🍽️:
- Chef → only cooks.
- Waiter → only serves food.
- Cashier → only handles money.

👉 If one person (say the waiter) is also forced to cook, serve, and handle money, the restaurant will be **chaotic**.
Each role should have **one responsibility**.

```java
class StudentData {
    public void storeData() {
        // save student data
    }
}

class GradeCalculator {
    public void calculateGrades() {
        // calculate grades
    }
}

class ReportPrinter {
    public void printReport() {
        // print student report
    }
}
```

⚠️ Problem:
This class has **too many jobs** → storing, calculating, printing.
Example: Suppose the university switches from **printing paper reports** 📝 to **sending PDF reports** 📧.
→ You must go into this Student class and change the printReport() method.
Suppose the database changes from **MySQL** to **MongoDB**.
→ You must go into this Student class again and change the storeData() method.
So this single class will keep changing for **different reasons**.

👉 Now:
- `StudentData` → only saves student info.
- `GradeCalculator` → only calculates grades.
- `ReportPrinter` → only prints report.

Each class has **one job = one reason to change** → easy to maintain, test, and extend.

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 = /^([\w-]+(?:\.[\w-]+)*)@((?:[\w-]+\.)*\w[\w-]{0,66})\.([a-z]{2,6}(
        return re.test(email);
    }
    greet() {
        alert("Hi!");
    }
}
```

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!");
    }
}
```

TypeScript is a superset of JavaScript that adds **static typing** to the language. It helps catch potential errors at compile-time rather than runtime, which improves code quality, maintainability, and developer experience.

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?

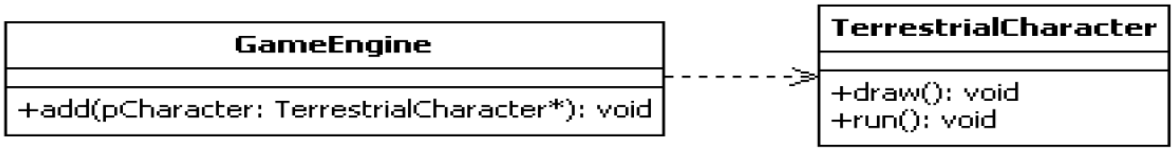
- Being close to modifications does not mean that designs cannot be modified; it means that modifications should be done by adding new code, and incorporating this new code in the system in ways that does not require old code to be changed!

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.*

لنفترض وجود نظام ألعاب خيالي يتضمن عدة أنواع من الشخصيات الأرضية، تلك التي يمكنها التجول بحرية على الأرض. ومن المتوقع إضافة شخصيات جديدة في المستقبل.



Extra:This shows that the `GameEngine` class takes an object of type `TerrestrialCharacter` as a parameter.

Because of this, `GameEngine` depends on `TerrestrialCharacter`—if you remove or change `TerrestrialCharacter`, `GameEngine` will be affected.

Why Violate??? The GameEngine class depends directly on the concrete class TerrestrialCharacter.

If you want to add new types of characters (e.g., AquaticCharacter, AlienCharacter), you would need to modify GameEngine's code to handle them.

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?

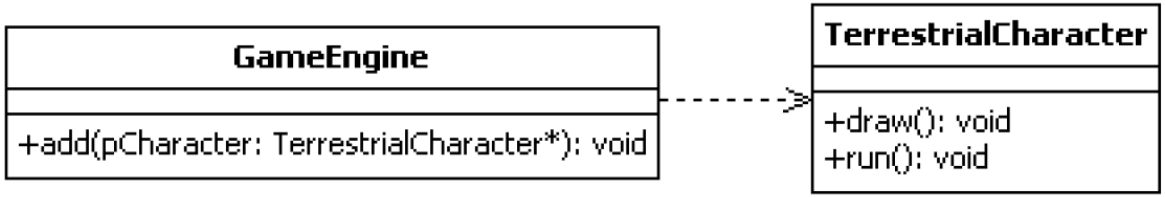
What happens if we add a new requirement to support other types of characters, e.g., an AerialCharacter that can fly?

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...

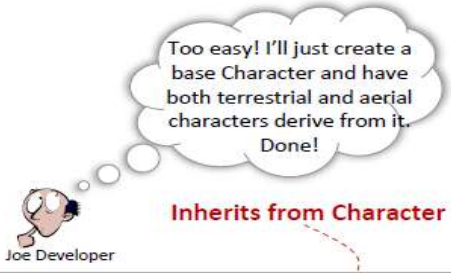
The Open-Close Principle (OCP)

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?



THE OPEN-CLOSED PRINCIPLE (OCP)




```
class Character {

public:
  // Get the type of character.
  virtual string getType() = 0;

  // Draw the character on the screen.
  virtual void draw() = 0;
};
```

Joe Developer decided to abstract the Character concept and separate it from more specific Character types

Inherits from Character

```
class AerialCharacter : public Character {

public:
  // Get the type of character.
  virtual string getType() {

    // Return the type of character.
    return "aerial";
  }

  // Draw the character on the screen.
  virtual void draw() {

    // Code to draw the aerial character.
    cout<<"drawing aerial character!\n";
  }

  // Make the character fly!
  virtual void fly() {

    // Code to make the character fly.
    cout<<"character flying!\n";
  }
};
```

Since Terrestrial characters run and Aerial ones fly, Joe decided to delegate creation of these functions to subtypes, namely, TerrestrialCharacter and AerialCharacter

Are we done? Not really! The getType(...) function should give you an indication why we're still violating the OCP. Let's take a closer look in the next slide...

Inherits from Character

```
class TerrestrialCharacter : public Character {

public:
  // Get the type of character.
  virtual string getType() {

    // Return the type of character.
    return "terrestrial";
  }

  // Draw the character on the screen.
  virtual void draw() {

    // Code to draw the terrestrial character.
    cout<<"drawing terrestrial character!\n";
  }

  // Make the character run!
  virtual void run() {

    // Code to make the character run.
    cout<<"character running!\n";
  }
};
```

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

The Open-Closed Principle (OCP)

Design Principle: Encapsulate Variation

```
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!

Notice how the GameEngin client needs to know the ty of Character before it can activate it. This is a side-eff of a violation of the OCP



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...

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! */ }
};
```

```
// The terrestrial character.
class TerrestialCharacter : 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! */ }
};
```

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

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

Extra:

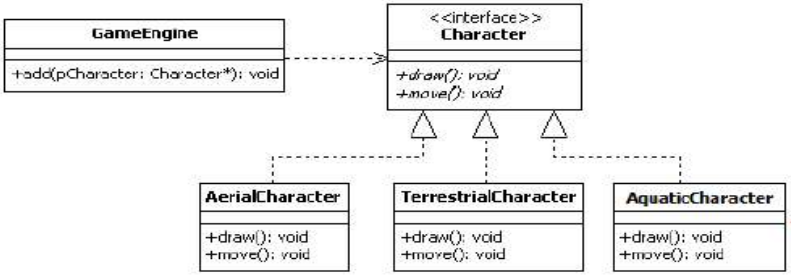
```java
// 1) Abstraction
public interface Character {
    void draw();
    void move(); }  // generalized action (run, fly, swim, etc.)
// 2) Existing characters
public class TerrestrialCharacter implements Character {
    @Override
    public void draw() {
        System.out.println("Drawing terrestrial character...");}
    @Override
    public void move() {
        System.out.println("Running on the ground..."); }}

public class AerialCharacter implements Character {
    @Override
    public void draw() {
        System.out.println("Drawing aerial character..."); }
    @Override
    public void move() {
        System.out.println("Flying in the sky..."); }}

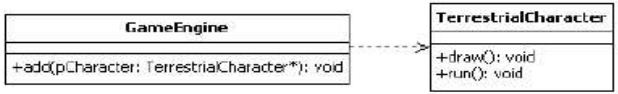
// 4) Game engine
public class GameEngine {
    public void add(Character character) {
        // Activate character immediately when added
        character.draw();
        character.move(); }}
// 5) Usage
public class Main {
    public static void main(String[] args) {
        GameEngine engine = new GameEngine();
        // Add characters (engine code unchanged)
        engine.add(new TerrestrialCharacter());
        engine.add(new AerialCharacter());}}
```

THE OPEN-CLOSED PRINCIPLE (OCP)

New redesign! Adheres to OCP!



Old design! Violates OCP!



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

```
// 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.
};
```

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

```java
// 1) Abstraction
public interface Character {
    void draw();
    void move(); }  // generalized action (run, fly, swim, etc.)
// 2) Existing characters
public class TerrestrialCharacter implements Character {
    @Override
    public void draw() {
        System.out.println("Drawing terrestrial character...");}
    @Override
    public void move() {
        System.out.println("Running on the ground..."); }}

public class AerialCharacter implements Character {
    @Override
    public void draw() {
        System.out.println("Drawing aerial character..."); }
    @Override
    public void move() {
        System.out.println("Flying in the sky..."); }}

// 4) Game engine
public class GameEngine {
    public void add(Character character) {
        // Activate character immediately when added
        character.draw();
        character.move(); }}
// 5) Usage
public class Main {
    public static void main(String[] args) {
        GameEngine engine = new GameEngine();
        // Add characters (engine code unchanged)
        engine.add(new TerrestrialCharacter());
        engine.add(new AerialCharacter());}}
```

Extra:

```java
// 3) New character (extension, no engine changes!)
public class AquaticCharacter implements Character {
    @Override
    public void draw() {
        System.out.println("Drawing aquatic character...");
    }

    @Override
    public void move() {
        System.out.println("Swimming in the water...");
    }}
```

```java
engine.add(new AquaticCharacter());
```

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

Extra:

No design can completely avoid changes. In any software system, certain parts will eventually need to be modified.

The (OCP) encourages designing software so that the **parts likely to change** are identified early.

These changes can be managed by using **polymorphism(Overriding)**, which allows you to extend functionality **without changing existing code**, making it open for extension but closed for modification.

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 LSP requires:

- **Signatures** between base and derived classes to be maintained
- **Subtype specification** supports reasoning based on the super type specification

Extra:

•**Definition:** A subclass must be usable anywhere its parent class is expected, without breaking the program's correctness.

1. **Same method signatures** → Derived classes must not change the method's parameters or return type.
2. **Behavioral contract** → Subclasses should honor the rules/expectations set by the base class (e.g., if move() means "cause movement," the subclass shouldn't break that meaning).

Base design

```java
public interface Character {
    void draw();
    void move();
}
```

Subtypes

```java
public class TerrestrialCharacter implements Character {
    public void draw() { System.out.println("Drawing terrestrial..."); }
    public void move() { System.out.println("Running on the ground..."); }
}

public class AquaticCharacter implements Character {
    public void draw() { System.out.println("Drawing aquatic..."); }
    public void move() { System.out.println("Swimming in the water..."); }
```

GameEngine

```java
public class GameEngine {
    public void add(Character c) {
        c.draw();
        c.move();
    }
}
```

Extra:

❌ LSP violation

Imagine this bad class:

```java
public class GhostCharacter implements Character {
    @Override
    public void draw() { System.out.println("Drawing ghost..."); }

    @Override
    public void move() {
        throw new UnsupportedOperationException("Ghosts don't move!");
    }
}
```

Extra: Example

➢ The **Liskov Substitution Principle (LSP)** means that objects of a derived class (subclass) should be able to replace objects of the base class (superclass) without causing errors or changing the program's behavior. This ensures that the subclass correctly follows the behavior expected from the base class.

Base Class:

```typescript
class Bird {
  fly(): void {
    console.log("Flying");
  }
}
```

Violation of LSP:

```typescript
class Penguin extends Bird {
  fly(): void {
    throw new Error("Penguins can't fly!");
  }
}
```

Derived Class (Honoring LSP):

```typescript
class Sparrow extends Bird {
  fly(): void {
    console.log("Sparrow flying");
  }
}
```

Here, replacing Bird with Penguin would break the program because Penguin doesn't behave like other birds that can fly. This violates LSP because it changes the expected behavior of the base class (Bird).

In this case, the Sparrow class can replace the Bird class in the program without changing how it works, as both have the fly() method.

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.

Extra:

•The method signatures in the subtype must match those in the base class.

•This ensures that a program, if type-correct for the base class, will remain type-correct for the derived class. For example, the return types and parameter types must stay the same or be more general in the derived class.

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.

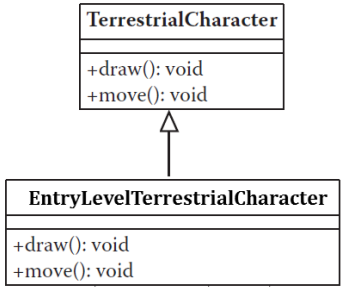
•This rule ensures that reasoning based on calls to base class methods remains valid when they are overridden in a subtype.

•Even though a method might be overridden in the subclass, it should still follow the base class contract and not change the expected behavior.

- **Subtype methods can weaken pre-conditions, not strengthen them (i.e., require less, not more).**
- **Subtype methods can strengthen post-conditions, not weaken them (i.e., provide more, not less).**

The Liskov Substitution Principle (LSP) - example

Does this example apply both OCP and LSP?



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() {

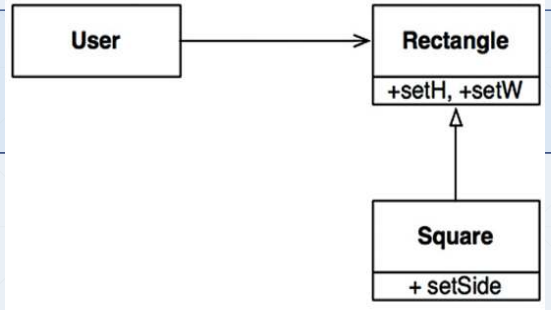
    // code to make the character fly.
  }
  // Code here to implement all other interfaces.
};
```

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.

titution Principle (LSP) – example 2

ion of the LSP is the famed square/rectangle problem

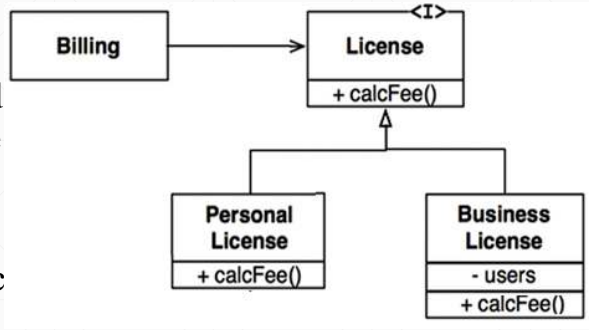
- In this example, Square is not a proper subtype of Rectangle because the height and width of the Rectangle are independently mutable;
- in contrast, the height and width of the Square must change together.
- Since the User believes it is communicating with a Rectangle, it could easily get confused.



A simple violation of substitutability, can cause a system's architecture to be polluted with a significant amount of extra mechanisms.

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 whic of the two subtypes it uses.
- Both of the subtypes are substitutable for the License type.



Does this example apply both OCP and LSP or not?

Extra

➢ Yes, this example applies both **Open/Closed Principle (OCP)** and **Liskov Substitution Principle (LSP)**. Here's how:

➢ **Open/Closed Principle (OCP)**:

- The **License** class is open for extension and closed for modification. You can extend the functionality by adding new license types (e.g., **PersonalLicense** and **BusinessLicense**) without modifying the existing **License** class. This shows that the system is **open for extension** while being **closed for modification**.

➢ **Liskov Substitution Principle (LSP)**:

- The **PersonalLicense** and **BusinessLicense** classes are subtypes of **License** and can be substituted in place of a **License** without affecting the behavior of the **Billing** class. This means that the **Billing** class does not need to know or depend on which specific subtype is being used, fulfilling the LSP.

Extra:

```java
// 1) Base interface (License)
public interface License {
    double calcFee(); }  // all licenses must calculate fee

// 2) Personal License (subtype)
public class PersonalLicense implements License {
    @Override
    public double calcFee() {
        // Simple rule for personal users
        return 100.0;   }}

// 3) Business License (subtype)
public class BusinessLicense implements License {

    @Override
    public double calcFee() {
        // Fee depends on number of users
        return users * 50.0;}}

// 4) Billing system (uses License, not subtypes directly)
public class Billing {
    public void printBill(License license) {
        System.out.println("License fee: " + license.calcFee());}}

// 5) Test
public class Main {
    public static void main(String[] args) {
        Billing billing = new Billing();

        License personal = new PersonalLicense();
        License business = new BusinessLicense(5);

        billing.printBill(personal);    // works with PersonalLicense
        billing.printBill(business);    // works with BusinessLicense
    }}
```

Lecture Outlines

- **Overview of Component Design**
- **Designing Internal Structure of Components (OO Strategy)**
- **Design Principles for Internal Component Design (SOLID)**
  - **S**ingle Responsibility Principle
  - **O**pen-Closed Principle
  - **L**iskov Substitution Principle
  - **I**nterface-Segregation Principle (Next lecture)
  - **D**ependency-Inversion Principle (next lecture)
- **Designing Internal Behavior of Components** (next lecture)