💎 Key Takeaways

✅ In-depth understanding of SOLID principles

✅ Walk-throughs with examples

✅ Understand concepts like Dependency Injection, Inversion of Control

✅ Practice quizzes & assignment

❓ FAQ

▶ Will the recording be available?
To Scaler students only

✏ Will these notes be available?
Yes. Published in the discord/telegram groups (link pinned in chat)

⏱ Timings for this session?
7.30pm - 10pm (2.5 hours) [15 min break midway]

🎧 Audio/Video issues
Disable Ad Blockers & VPN. Check your internet. Rejoin the session.

❔ Will Design Patterns, topic x/y/z be covered?
In upcoming masterclasses. Not in today's session.
Enroll for upcoming Masterclasses @ [scaler.com/events](https://www.scaler.com/events)

🖥 What programming language will be used?
The session will be language agnostic. I will write code in Java.
However, the concepts discussed will be applicable across languages

💡 Prerequisites?
Basics of Object Oriented Programming

👨‍💻 About the Instructor

Pragy
[linkedin.com/in/AgarwalPragy](https://www.linkedin.com/in/AgarwalPragy/)

Senior Software Engineer + Instructor @ Scaler

Important Points

💬 Communicate using the chat box

🙋 Post questions in the "Questions" tab

💙 Upvote others' question to increase visibility

👍 Use the thumbs-up/down buttons for continous feedback

⏱️ Bonus content at the end

>
> ❓ What % of your work time is spend writing new code?
>
> • 10-15%   • 15-40%   • 40-80%   • > 80%
>

< 15% of a dev's time is spent writing fresh code!

⏱️ Where does the rest of the time go?

- debugging, refactoring, testing
- documentations, meetings, reading code, stackoverflow
- mentoring, team acitvities
- break - chai sutta break

✅ Goals

Make sure that we work less and play more - whatever work you do is correct in the first go.

We'd like to make our code

1. readable
2. maintainable
3. testable
4. extensible

Robert C. Martin 👴🏻 Uncle Bob

====================

💎 SOLID Principles

- Single Responsibility
- Open-Closed
- Liskov's Substitution
- Interface Segregation
- Depdency Inversion

Interface Segregation / Inversion of Control
Dependency Inversion / Dependency Injection

💭 Context

- Game - Zoo application 🦊
- Animals, Staff, Visitors, Cages, Food, Shows, ..

🎨 Design a Zoo Entity

```java

// Pseudocode

// Object Oriented Programming
// entities/concepts => classes

class ZooEntity {
   // attributes - properties
   String species;          // animal
   Boolean hasWings;        // animal
   String name;             // staff
   String phoneNumber;      // staff
   String visitorPassId;    // visitor
   Integer enclosureWidth;  // cage
   String foodType;         // food
   DateTime startTime;      // show

   void chew() {
      ...
      // make changes here - effects both staff and animal behavior
   }

   void animalEat() {
      chew()
   }

   void staffEat() {
      chew()
   }

   // methods - behavior
   void eat();  // animals can eat
   void poop();
```

```
    void speak();

    void markAttendance();
    void getPaid();
    void cleanZoo();
    // staff can also eat

    void petAnimal();
    void getEatenByAnimalByBeingStupid();
    void giveFeedback();
    // visitors can also eat
}
```

🐞 Problems with the above code?

❓ Readable
unexperienced dev: I can read it, and I can understand it
(thinking about now)

experienced dev: As the complexity grows, and the number of entities increase, this class will become unreadable
(thinking about the future)

❓ Testable
unexperienced dev: I can write testcases for each method
experienced dev: I can write testcases, but the code will be coupled - changes to one of code will effect other pieces of code

❓ Extensible
come back to this

❓ Maintainable
merge conflicts - because we have multiple devs working on the same file

🛠️ How to fix this?

====================================
⭐ **Single Responsibility Principle**
====================================

- Every function/class/package/module (unit of code) should have only 1 well-defined responsibility

- any piece of code has just 1 reason to change

- if a piece of code has multiple responsibilities, then we should break it down into multiple pieces of code

```java
class ZooEntity {
    String id;
    Datetime createdAt, updatedAt;
}

class Animal extends ZooEntity {
    String species;           // animal
    Boolean hasWings;         // animal

    void eat();  // animals can eat
    void poop();
    void speak();
}


class Staff extends ZooEntity {
    String name;              // staff
    String phoneNumber;       // staff

    void markAttendance();
    void getPaid();
    void cleanZoo();
}


class Visitor extends ZooEntity { ... }
class Show extends ZooEntity { ... }
```

- Readable
We have a lot more classes now - number of classes has increased
non-issue - you can solve that by using metaprogramming (generics, templates, macros, preprocessor, higher order functions)

Readability is not about how much code there is - because as a dev you will be working with a small part of the codebase

- Testable
Yes, we've improved, because now, changes made to one class will NOT effect the other classes

- Maintainable
merge conflicts will reduce

```java
class Animal {
    // different animals will behave differently
    // birds are different from fishes, ...
}
```

```java

class Animal extends ZooEntity {
    String species;
    String name;
}

class Bird extends Animal {
    void fly();
    void layEgg();
}
```

```java
class Mammal extends Animal {
    void keepWarm();
}
```

🐦 Design Birds

```java

class Bird extends Animal {
    void peck() { ... }

    void fly() { ... }
}

```

🕊️ Different birds fly differently

```java
class Bird extends Animal {
    void fly() {

        // decided the flying behavior based on the species

        if (species == "sparrow")
            print("flap wings & fly low")
        else if (species == "eagle")
            print("soar high")
        else if (species == "peacock")
            print("only the females fly, males cant")

    }
}
```

🐞 Problems with the above code?

- Readable
- Testable
- Maintainable

- Extensible - FOCUS!

Do you always write code from scratch? OR do you use other people's code (importing)

```java
[library] package SimpleZooLibary {
    // .dll .com .so .jar
    class Bird extends Animal {
        void fly() {

            // decided the flying behavior based on the species

            if (species == "sparrow")
                print("flap wings & fly low")
            else if (species == "eagle")
                print("soar high")
            else if (species == "peacock")
                print("only the females fly, males cant")

        }
    }
}

[executable] Game {

    import SimpleZooLibary.Bird;

    // imagine if we wanted to add new types of Birds
    // it will be painful to extend the library code here

    class ZooGame {
        static void main(String args[]) {
            Bird b = new Bird(...);
        }
    }
}
```

Do we always have the permissions to modify a library that we've imported?
No
Closed sourced - we don't even have the code
Open sourced - we can see the code, but not necessarily modify it

Zerodha - largest stock-trading platforms for India
Kite API - can be used to implement your own trading algorithms

```java
class Trader {
    void executeTrade() {
        if(strategy == "mean-reversion") {
            ...
        } else if (strategy == "fibonacci-retracement") {
            ...
        }
    }
}
```
Anyone using the Kite API is restricted to only strategies that are pre-coded by Zerodha

🛠 How to fix this?

========================
⭐ **Open-Closed Principle**
========================

- Code should be open for extension, yet closed for modification!

? Why closed for modification

1. dev - write code on local machine, test it, make a PR
2. PR goes for review - team members will suggest changes (back to step 1)
   PR will be merged
3. the code goes to the QA team - ensure that everything is working. Add new testcases. Integrations tests
4. CI/CD
    + staging servers (production like environment) - everything is working fine
    + production servers
        * A/B testing
            - deploy the code/feature for 5% of the userbase
            - test - exception, performance, revnue, NPS ...
        * deployed to 100% of the codebase

```java
[library] package SimpleZooLibary {
   // .dll .com .so .jar

   abstract class Bird extends Animal {
      abstract void fly();
   }

   class Sparrow extends Bird {
      void fly() { print("flap wings and fly low"); }
   }

   class Eagle extends Bird {
      void fly() { print("soar high"); }
   }
}

[executable] Game {

   import SimpleZooLibary.Bird;

   // imagine if we wanted to add new types of Birds
   // it will be painful to extend the library code here
   class Peacock extends Bird {
      ... implement custom functionality here
   }

   class ZooGame {
      static void main(String args[]) {
         Bird b = new Bird(...);
      }
   }
}
```

```java
abstract class TradingStrategy {
   Trade getTrade();
}

class Trader {
   void executeTrade(TradingStrategy strategy) {
      Trade trade = strategy.getTrade();
      trade.execute();
   }
}
```

```java
class MeanReversionTradingStrategy extends TradingStrategy {
    ...
}
```

- Extension
A developer who doesn't have modification permissions for the codebase can still import the code and extend the functionality!

Software salaries in India - upto 3Cr. per annum (recurring with stocks)
Why would a company pay you this amount?
   Because you're able to anticipate future requirements and account for them today itself!

It is very easy to be a dev, but it is very hard to be a good dev
There's just so much to know

Tier-1 company (MAANG)

Interview process
1. screening round (HR)
   - what projects have you worked on
   - estimate the value of 2^64 without using a calculator
      - 2^10 ~ 10^3
      - 2^60 ~ 10^18
      - 2^64 ~ 16 * 10^18 (a little more than 16 quadrillion)
2. telephonic coding interview / online coding round
   - Data Structures & Algorithms (DSA)
3. On-site interviews
   a) DSA round
   b) SDE-1 => SQL, Low Level Design (SOLID principles, Design Patterns, schema design, class diagram)
      SDE-2 => (everything SDE-1) + concurrency, advanced SQL queries, query optimization, REST API design
      SDE-2+ => High Level Design

There's tons of resources to learn all these things
   - ping me on Linkedin
   - join our whatsapp/discord/telegram communities

Just learning/reading is not enough - practice is king!

Scaler is a 11 month long, very rigourous and in-depth, curriculum
Soft-skills - resume building, salary negoations, .. mentor sessions

How do you measure success?
If you can't measure it, you can't optimize it!

How we meausre success @ Scaler?
Look at the salary before Scaler, and the Salary after Scaler

8LPA salary before Scaler.
After 11 months of Scaler, if they're able to get a salary of 30LPA

30/8 = 3.75x
gain/delta/impact = 2.75x

Average impact across all Scaler learners is 2.6x

8.53 - ~12 min break
resume the class sharp at 9.05

Single Responsibility Principle
Open-Closed Principle

🐓 Can all birds fly?

```java

abstract class Bird {
    abstract void fly();
}

class Sparrow extends Bird { void fly() { print("fly low"); }}
class Eagle extends Bird { void fly() { print("soar high"); }}

class Kiwi extends Bird {
    void fly() {
        // ... what to do here?
    }
}

```

Emu, Kiwi, Ostrich, Penguin, Dodo, Flamingo .. are all examples of birds which cannot fly

>
> ❓ How do we solve this?
>
> • Throw exception with a proper message
> • Don't implement the `fly()` method
> • Return `null`
> • Redesign the system
>

🏃 Try the quickest/simplest solution first

Don't implement the `fly()` method at all!
```java
abstract class Bird {                // blueprints - incomplete concepts
    abstract void fly();
}

class Kiwi extends Bird {
    // no void fly()
}
```

🐞 Compiler Error
You marked the class Bird as `abstract` - you told me that it is incomplete.
You said that you don't know how to implement `fly` inside Bird, and that whoever extends this will supply the implementation

Either provide implementation of `void fly()` inside `class Kiwi extends Bird` or you should mark the class Kiwi as abstract too!

⚠ Throw a proper exception

```java
abstract class Bird {                // blueprints - incomplete concepts
    abstract void fly();
}

class Kiwi extends Bird {
    void fly() {
        throw new FlightlessBirdException("Kiwi is a flightless species of Bird");
    }
}
```

🐞 Wrong - this violates expectations!

```java
abstract class Bird {
    abstract void fly();
}

class Sparrow extends Bird { void fly() { print("fly low"); }}
class Eagle extends Bird { void fly() { print("soar high"); }}


class Game {
    Bird getBirdFromUserChoice() {
        // input bird type from User
        // return an object of the appropriate class
    }

    void main() {
        Bird b = getBirdFromUserChoice();
        b.fly();
    }
}



// extension to the code
class Kiwi extends Bird {
    void fly() {
        throw new FlightlessBirdException("...");
    }
}
```

✅ Before extension
code works fine - everyone is happy

❌ After extension
did we modify the existing code? No
do we necessarily know who/when the extension was added? in a large team, No

suddenly, without modifying existing code, the existing code now breaks!

==================================

⭐ Liskov's Substitution Principle

==================================

- child classes should honor the contract made by the parent classes

- don't violate expectations set by the parent class

🎨 Redesign the system

```java
abstract class Bird {
   abstract void eat();
   abstract void poop();
}

interface ICanFly {  // interface - prefix the interface name with I
   void fly();
}

class Sparrow extends Bird implements ICanFly {
    void fly(){ print("fly low"); }
}
class Eagle extends Bird implements ICanFly {
    void fly() { print("soar high"); }
}

class Game {
   Bird getBirdFromUserChoice() {
      // input bird type from User
      // return an object of the appropriate class
   }

   void main() {
      Bird b = getBirdFromUserChoice();
      if(b instanceof ICanFly) {
         ICanFly flyingBird = (ICanFly) b;
         flyingBird.fly();
      }
   }
}

// extension to the code
class Kiwi extends Bird {
   // don't have to implement void fly because it doesn't implement the ICanFly 
interface!
}
```

1. why not use strategy pattern here?
2. why not simply add a FlightlessBird class as a class hierarchy?
3. why not have composition here over inheritance?
4. why not have a boolean canFly inside Bird class?

✈ What else can fly?

```java

interface ICanFly {
    void fly();

    // what does a Bird do before it starts flying?
    // spread wings, jump a little to take off

    void smallJump();
    void spreadWings();
    void flapWings();
}


class Shaktiman implements ICanFly {
    void fly() { print("rotate very fast"); }

    void flapWings() {
        // Sorry Shakltiman :(
    }
}

```

Are there things apart from Birds which can fly?

Insects / Aeroplanes / Poop / Paper rockets / Shaktiman / Udta Punjab

>
> ❓ Should these additional methods be part of the ICanFly interface?
>
> • Yes, obviously. All things methods are related to flying
> • Nope. [send your reason in the chat]
>

No.

⭐ Interface Segregation Principle

- Keep your interfaces minimal
- A class should not implement an interface if it doesn't use ALL of its methods

Java - Interfaces vs Abstract classes - use interfaces if you don't have any state
Python - only abstract class + multiple inheritance
C++ - pure virtual methods
Rust/Scala - traits

How will you fix `ICanFly`?

```java

abstract class Animal {
    abstract void eat();
    abstract void poop();
}

abstract class Bird extends Animal {
    
}

interface ICanFly {
    void fly();
}

interface IHasWings {
    void spreadWings();
    void flapWings();
}

interface ICanJump {
    void smallJump();    
}

class Sparrow extends Bird implements ICanFly, IHasWings, ICanJump {
    void fly() {...}
    void spreadWings() {...}
    void flapWings() {...}
    void smallJump() {...}
}


class Shaktiman extends Animal implements ICanFly, ICanJump {
    void eat() {...}
    void poop() {...}
    void fly() {...}
    void smallJump() {...}
}

```

Low-level design

We've designed the creatures in the zoo - let's design the structures

🗑️ Design a Cage

```java

// what things does a Cage "depend" on?
// door - to prevent attacks and stop animals
// bowl - to feed animals
// animal - the inhabitants

interface IBowl { void feed(Animal animal); }       // high level - abstract
class MeatBowl implements IBowl {
   public MeatBowl() {
      // acquire meat
      // tenderize it                               // low level - details
      // cook it
      // add enzymes
      // add preservatives
   }

   void feed(Animal animal) {
      // check for diet plan
      // check for allergies
      // check for dental health
      // is the animal stray or part of zoo
      // feed
   }
}
class FruitBowl implements IBowl { ... }
class GrainBowl implements IBowl { ... }

interface IDoor { void handleAttack(Attack attack); }
class IronDoor implement IDoor { ... }
class WoodenDoor implement IDoor { ... }
class AdamantiumDoor implement IDoor { ... }

abstract class Animal { ... }
class Tiger extends Animal { ... }
abstract class Bird extends Animal { ... }
class Peacock extends Bird { ... }
class Sparrow extends Bird { ... }

class Cage1 {
   // Cage1 is a cage for birds

   FruitBowl bowl = new FruitBowl(...);
   WoodenDoor door = new WoodenDoor(...);
   List<Bird> birds = Arrays.asList(new Peacock(...),
                                    new Sparrow(...))

   public Cage1() {
      ...
   }

   void handleAttack(Attack attack) {
      // delegating to the appropriate dependency
      this.door.handleAttack(attack);
   }

   void feed() {
      // delegate to the bowl
      for(Bird b: this.birds) {
         this.bowl.feed(b);
      }
   }
}

class Cage2 {
   // Cage2 is for big cats - tigers, lions, ...

   MeatBowl bowl = new MeatBowl(...);
   IronDoor door = new IronDoor(...);
   List<Cat> cats = Arrays.asList(new Tiger(...),
                                    new Lion(...))

   public Cage2() {
      ...
   }

   void handleAttack(Attack attack) {
      // delegating to the appropriate dependency
      this.door.handleAttack(attack);
   }

   void feed() {
      // delegate to the bowl
      for(Cat c: this.cats) {
         this.bowl.feed(c);
      }
   }
}

// add another cage for X-men
// 1. create a new class Cage3
// 2. initialize the dependencies - MeatBowl, AdamantiumDoor, ...

class Game {
   void main() {
      Cage1 forBirds = new Cage1();
      Cage2 forCats = new Cage2();
   }
}
```

🐞 Lot of code repetition - true

Let's look at this from a dependency diagram perspective

- High level code: code that gives you the overview, without delving into the details. They just tell you what methods are there, without telling you the implementation of the methods
    - interfaces/abstract classes
    - Managerial/Controller code - doesn't do things itself - it delegates tasks to other things
- Low level code: implementation details - concrete classes/functions - tell exactly how the code works

```
  -------         ---------        -------
   IBowl           IAnimal          IDoor            high level
  -------         ---------        -------
     ║               ║                ║
┌──────────┐      ┌───────┐     ┌──────────┐
│ MeatBowl │      │ Tiger │     │ IronDoor │         low level
└──────────┘      └───────┘     └──────────┘
     │               │                │
     └───────────────┼────────────────┘
                     │
```

```
                 Cage2                     controller - high level
```

high level unit `Cage2` depends on low-level details `MeatBowl`, ...

=================================
⭐ **Dependency Inversion Principle**   **- what to do**
=================================

- High-level code should NEVER depend on low level details
- Instead, it should depend on high level abstractions

```
   -------       ---------       -------
    IBowl         IAnimal         IDoor            high level
   -------       ---------       -------
      |              |              |
      ----------------------------------
                     |
                   Cage                            high level
```

But how?

=======================
💉 **Dependency Injection**   **- how to do it**
=======================

- instead of creating & managing our own dependencies, we let our clients provide the dependencies for us

```java

class Cage {
   // generic cage - for any sort of animal

   IBowl bowl;   // depending on abstraction
   IDoor door;   // I don't care exactly which type of bowl I need
   List<Animal> animals;

   // injected the dependencies via the constrcutor
   //                 vvvv        vvvv                vvvvvvv
   public Cage(IBowl bowl, IDoor door, List<Animal> animals) {
      this.bowl = bowl;
      this.door = door;
      this.animals.addAll(animals);
   }
}

class CustomCage extends Cage {
   IVisitorArea visitorArea;
}

class Game {
   void main() {
      Cage forBirds = new Cage(
         new FruitBowl(...),
         new WoodenDoor(...),
         Arrays.asList(new Sparrow(...), new Peacock(...), ..)
      );

      Cage forCats = new Cage(
         new MeatBowl(...),
         new IronDoor(...),
         Arrays.asList(new Tiger(...), ...)
      );

      Cage forXMen = new Cage(
         new MeatBowl(...),
         new AdamantiumDoor(...),
         Arrays.asList(new Wolverine(...), new Cyclops(...))
      );
   }
}
```

Enterprise Code
================

1. getting your foot into the door - cracking the interviews
2. surviving & thriving once you get in

Large companies like Google/Amazon - you will find "over-engineered" code

1. find obscure design patterns everywhere
   - singletons, factories, proxies, adapters, strategies
2. a ton of logging constructors
3. a ton of very ver long names `class RazorPayPaymentGatewayStrategy implements IPaymentGatewayStrategy { ... }`

Don't know Low Level Design
   1. you will be confused
   2. frustrated
   3. not be productive
   4. smallest of changes will take forever
If you know Low Level Design
   1. you don't even have to read the code most of the time
   2. just by looking at the filename / pattern name - you know exactly what the code will be doing!

================
🎁 **Bonus Content**
================

>
> We all need people who will give us feedback.
> That’s how we improve. 💬 Bill Gates
>

🧩 **Assignment**
----

https://github.com/kshitijmishra23/low-level-design-concepts/tree/master/src/oops/SOLID/

⭐ **Interview Questions**

> ❓ Which of the following is an example of breaking
> Dependency Inversion Principle?
>
> A) A high-level module that depends on a low-level module
> through an interface
>
> B) A high-level module that depends on a low-level module directly
>
> C) A low-level module that depends on a high-level module
> through an interface
>
> D) A low-level module that depends on a high-level module directly
>

> ❓ What is the main goal of the Interface Segregation Principle?
>
> A) To ensure that a class only needs to implement methods that are
> actually required by its client
>
> B) To ensure that a class can be reused without any issues
>
> C) To ensure that a class can be extended without modifying its source code
>
> D) To ensure that a class can be tested without any issues

>
> ❓ Which of the following is an example of breaking
> Liskov Substitution Principle?
>
> A) A subclass that overrides a method of its superclass and changes

```
>    its signature
>
> B) A subclass that adds new methods
>
> C) A subclass that can be used in place of its superclass without
>    any issues
>
> D) A subclass that can be reused without any issues
>




> ❓  How can we achieve the Interface Segregation Principle in our classes?
>
> A) By creating multiple interfaces for different groups of clients
> B) By creating one large interface for all clients
> C) By creating one small interface for all clients
> D) By creating one interface for each class



> ❓  Which SOLID principle states that a subclass should be able to replace
> its superclass without altering the correctness of the program?
>
> A) Single Responsibility Principle
> B) Open-Close Principle
> C) Liskov Substitution Principle
> D) Interface Segregation Principle
>



>
> ❓  How can we achieve the Open-Close Principle in our classes?
>
> A) By using inheritance
> B) By using composition
> C) By using polymorphism
> D) All of the above
>




# ============================ That's all, folks! ===========================
```