💎 Key Takeaways

✅ In-depth understanding of SOLID principles
✅ Walk-throughs with examples
✅ 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?
5pm - 8pm (3 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 devs time is spent writing fresh/new code

⏱️ Where does the rest of the time go?
- reviewing other people's code
  - reading
  - maintaining
  - bug fixing
- designing, thinking, discussing, RnD
- meeting
- docs / chatGPT / stackoverflow
- breaks - chai, TT, snooker

✅ **Goals**
========

Minimize work time and maximize play time
- only way to do this is to ensure that my work is VERY high quality

What you can't measure, you can't improve!

We'd like to make our code

1. Readability
2. Maintainability
3. Testability
4. Extensibility

Robert C. Martin 👨🏻 Uncle Bob

==========
💎 **SOLID Principles**

==========

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

S =/= Scalability / Substitution
S = Single Responsibility

I =/= Inversion of Control

D =/= Dependency Injection
D = Dependency Inversion

💭 Context

==========

- Simple Zoo Game 🦊
- characters - animals / birds / visitors / staff
- structures - cages / doors / food

My code will "look like" Java - it will actually be pseudo code
However, the concepts that we cover will be applicable to any language that supports OOP (C++/C#/Python/Typescript/Ruby/...)

🎨 Design a Character

==========

```java
class ZooEntity {
    // visitors / animals / zoo staff   -  concepts / ideas

    // attributes (properties)

    // ========= visitors ========
    String visitorName;
    Integer visitorAge;
    String address;
    String phone;
    String identityCard;
    // ========= animals ========
    String species;     // null for a visitor
    String animalName;
    Integer age;
    Color color;
    Integer weight;
    Gender gender;
    // ========= staff ========
    String staffName;   // null for a visitor
    Integer age;
```

```
    String address;
    String phone;
    String identityCard;

    // methods (behavior)

    // ========= visitors ========
    void walk();
    void feedAnimal();
    void speak();
    void eat();
    void poop();
    void payForTicket();
    // ========= animals ========
    void walk();
    void speak();
    void eat();
    void poop();
    void attack();
    void sleep();
    // ========= staff ========
    void walk();
    void feedAnimal();
    void speak();
    void eat();
    void poop();
}


class TestZooEntity {
    void testWalkForAnimal() {
        ZooEntity obj = new ZooEntity(...);
        ...
    }
}



```

🐞 Problems with the above code?

❓ Readable
1st look - code is readable - I can read it, and it makes sense!
But as the logic gets more complex, and the types of entities increase, this class ZooEntity will get extremely complex

To be able to read this class, I must understand the attributes and behavior of EVERY type of character in the zoo

❓ Testable
1st look - I can totally write testcases for each and every behavior

But, as the logic gets complex, the testcases will be tightly coupled - changing the behavior of Visitor can effect the testcases for Animal / Staff
That makes testing harder

❓ Extensible
We'll look at this later

❓ Maintainable
Merge conflicts - because multiple devs are working on the same file
(Git / other Version Control systems)

🛠️ How to fix this?

==========
⭐ **Single Responsibility Principle (SRP)**
==========

- Any class/function/package/module (any unit-of-code) should have a SINGLE, WELL-DEFINED responsibility

- any piece of code should have only 1 reason to change

- if some piece of code violates SRP - we should split it into multiple units

Prerequisite for today's class: Object Oriented Programming

```java
class ZooEntity { // shared attributes and behavior
   String name;
   Integer age;

   void walk();
   void speak();
   void eat();
   void poop();
}

class Animal extends ZooEntity {
   String species;
   Color color;
   Integer weight;
   Gender gender;

   void attack();
   void sleep();
}

class Visitor extends ZooEntity {
   String address;
   String phone;
   String identityCard;

   void feedAnimal();
   void payForTicket();
}

class TestVisitor {
   void testWalk() {
      ...
   }
}
```

```java
class Staff extends ZooEntity {
   String address;
   String phone;
   String identityCard;

   void feedAnimal();
}
```

Did we make any improvement on the previous metrics?

- Readable
Aren't there too many classes & files now?
As a dev (senior/junior/whatever) you will only be working on 1 or few features at a time - you're only looking at a few files at a time.
Each individual file is tiny and much much more readable!

- Testable
It is NOT possible for a change in the Visitor class to affect the testcases of other class - because they're separate classes that don't depend on each other

- Maintainable
If diff devs are working on diff classes, the number of merge conflicts will drastically reduce!

🪼 Design a Bird

```java
class Bird extends Animal {
   // String species; - inherited from the parent class Animal

   Integer wingSpan;
   Boolean hasBeak;

   void fly() {
      print("flap wings")
   }

}
```

🕊️ Different birds fly differently!

```java
class Bird extends Animal {
   // String species; - inherited from the parent class Animal

   Integer wingSpan;
   Boolean hasBeak;
```

```
    void fly() {
        // if-else ladder

        if(species == "Sparrow")
            print("fly low")
        else if(species == "Pigeon")
            print("fly overhead and poop bomb people")
        else if(species == "Eagle")
            print("glide elegantly high above")
        else
            print("flap wings")
    }
}
```

🐞 Problems with the above code?

- Readable
- Testable
- Maintainable

- Extensible - FOCUS!

Do you always implement all code from scratch, or do you use external libraries often?

With respect to external libraries,
- do you have permission to modify the code of the external libary? NO
- you might not even have access to the code
    - libraries are shipped as compiled files (.dll, .com, .exe, .jar, .class, .pyc, .egg, .so)

🛠️ How to fix this?

==========
⭐ **Open/Closed Principle (OCP)**
==========

- Your code should be closed for modification, but still, open for extension!

❓ Why should code be closed for modification? What's so bad about modifying working code?

Lifecycle of code
- dev: implement & test code on local machine.
- dev: commit & push
- Pull Request - reviewed by others in the team - feedback - iterate
- Pull Request - merged
- Go for deployment
    + Quality Assurance (QA) team - write more testcases (unit/integration)
    + Staging servers
      + look at logs and see if there is any issue
    + Deploy to users

```
        * A/B testing
            - deploy to only 5% of the users
            - check metrics
                - num of exceptions
                - user ratings
                - revenue
                - load on machines
        * Finally code is deployed to all users
```

```java
[external library] SimpleZooLibaryFromGithub {
   class Animal { ... }
   class Bird extends Animal {
      // String species; - inherited from the parent class Animal

      Integer wingSpan;
      Boolean hasBeak;

      void fly() {
         // if-else ladder

         if(species == "Sparrow")
            print("fly low")
         else if(species == "Pigeon")
            print("fly overhead and poop bomb people")
         else if(species == "Eagle")
            print("glide elegantly high above")
         // else if (species == "Peacock")
         //    print("...")
         else
            print("flap wings")
      }
   }
}

[our code] ZooGame {
   import SimpleZooLibaryFromGithub.Animal;
   import SimpleZooLibaryFromGithub.Bird;

   // if I wish to add a new type of bird here - peacock,
   // how do I accomplish that?

   class ZooGame {
      void main() {
         Bird b = new Bird(species="Sparrow");
         b.fly();
      }
   }
}
```

Because the library author has violated Open/Close principle, it makes it difficult for the end user (who is also a dev) to extend the library code

In a large company, the "users" of your code might be other devs in other teams of the same company
Zerodha - Kite API - can be used to devs all across the world to implement trading bots

```java
[external library] SimpleZooLibaryFromGithub {
   class Animal { ... }
```

```
   abstract class Bird extends Animal {
      // String species; - inherited from the parent class Animal

      Integer wingSpan;
      Boolean hasBeak;

      abstract void fly();
   }

   class Sparrow extends Bird {
      void fly() { print("fly low") }
   }
   class Pigeon extends Bird {
      void fly() { print("fly overhead and poop bomb people") }
   }
   class Eagle extends Bird {
      void fly() { print("glide elegantly high above") }
   }
}

[our code] ZooGame {
   import SimpleZooLibaryFromGithub.Animal;
   import SimpleZooLibaryFromGithub.Bird;
   import SimpleZooLibaryFromGithub.Sparrow;
   import SimpleZooLibaryFromGithub.Eagle;

   // if I wish to add a new type of bird here - peacock,
   // how do I accomplish that?

   class Peacock extends Bird {
      void fly() { print("females can fly, males cant") }
   }

   class ZooGame {
      void main() {
         Bird b = new Sparrow();
         b.fly();
      }
   }
}
```

- Extension

That people who do NOT have modification access to your code should still be able to add functionality to the things that you've designed!
Your classes should be desinged with future inheritance in mind!

❓ Isn't this the same thing that we did for Single Responsibility as well?
Yes! All we did was take a large class and split it into multiple classes

❓ Does that mean that OCP == SRP?
No! The solution was the same, but the intent was different

🔗 All the SOLID principles are tightly linked together
If you adhere to 1 principle, very often you will get the other principles for free!

A lot of people are struggling with basic OOP concepts

It is important to not just know the concepts, but also, to know WHY these concepts exist in the first place

Assuming that you're already good at Data Structures and Algorithms, for getting into a senior role

Low Level Design

- Object Oriented Programming
  - interfaces / classes
  - abstraction / encapsulation / generalization
  - polymprphism / compile-time vs runtime polymoprphism
  - inheritance / composition
  - composition over inheritance
- SOLID principles
- Design Patterns
  - Creational (builder/singleton/factory)
    - understand the WHY of these patterns as well
    - for example, you should NEVER use the builder pattern in Python
    - builder pattern - workaround for missing features about object creation
      - named params
      - optional params
      - changing the order of params
      - validation
  - Behavioral
  - Structural
- Entity Relationship (database schema, and the class design)
- Case studies
  - TicTacToe / Snake Ladder / Chess
  - Parking Lot
  - Library Management
  - Splitwise
  - Bookmyshow (concurrency)

6.30 - resume by 6.45 sharp
in the meanwhile, if you have any question regarding career, please talk to Apurva

---

🐓 Can all the birds fly?

```java
class ZooEntity { ... }
class Staff extends ZooEntity { ... }
class Animal extends ZooEntity { ... }

abstract class Bird extends Animal {
   // String species; - inherited from the parent class Animal

   Integer wingSpan;
   Boolean hasBeak;

   abstract void fly();
}
```

```
class Sparrow extends Bird {
   void fly() { print("fly low") }
}
class Pigeon extends Bird {
   void fly() { print("fly overhead and poop bomb people") }
}
class Eagle extends Bird {
   void fly() { print("glide elegantly high above") }
}


class Penguin extends Bird {
   void fly() {
      throw new FlightlessBirdException("I'm a penguin bro, I can't fly!")
      return null
   }
}
```

Are there some birds which can't fly?
Penguins / Kiwis / Ostritch / Dodo / Emu / ...

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

🏃 Run away from the problem - Don't implement the `void fly()`

```java
class Penguin extends Bird {
   // no void fly here
}
```

🐞 Compiler Error - Bird is an abstract class - it has a missing function - abstract void fly. If Penguin inherits from Bird, either Penguin should supply the implementation of void fly, or Penguin itself will be incomplete (abstract)

class: blueprint
abstract class: incomplete blueprint
you cannot instantiate an abstract class

⚠ Throw a proper exception

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

```
        throw new FlightlessBirdException("Bro, don't you know Penguins can't fly");
    }
}
```

🐞 Violates Expectations!

```java
abstract class Bird extends Animal {
    abstract void fly();          // set expectation - Bird MUST fly
}

class Sparrow extends Bird {
    void fly() { print("fly low") }
}
class Pigeon extends Bird {
    void fly() { print("fly overhead and poop bomb people") }
}
class Eagle extends Bird {
    void fly() { print("glide elegantly high above") }
}


class ZooGame {
    Bird getBirdFromUserChoice() {
        // show the different species to the user
        // when user makes a selection, create an object of that type of bird
        // return this object
    }

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


// INTERN - told to work on Kiwi class

class Kiwi extends Bird {
    void fly() {
        throw new FlightlessBirdException();    // - violating expectation
    }
}
```

✅ Before extension
Code works, it is well tested, dev is happy, user is happy!

❌ After extension
1. Did we change any of the existing code? NO
2. Was the existing code working earlier?   YES
3. So, logically, should the existing code continue working?   YES, of course!
4. Is it?

==========
⭐ **Liskov's Substitution Principle**
==========

- Do NOT violate expectations set by your parent class

- Any type should be replacable by its subtype
- Anywhere you use an object of `class Parent`, you should also be able to use an object of `class Child extends Parent`
- You should be able to sustitute the Child class in place of the Parent class without breaking any code!

🎨 Redesign the system!

```java

abstract class Bird {
   Boolean hasBeak;
   Integer wingSpan;

   // since we know that not all birds can fly,
   // we should NOT have the abstract void fly here

   // abstract void peck(); // I know that every bird can peck
}

interface ICanFly {
   void fly();
}


class Sparrow extends Bird implements ICanFly {
   void fly() { print("fly low") }
}
class Pigeon extends Bird implements ICanFly {
   void fly() { print("fly overhead and poop bomb people") }
}
class Eagle extends Bird implements ICanFly {
   void fly() { print("glide elegantly high above") }
}



class Kiwi extends Bird {
   // note that since Kiwi doesn't implement the interface ICanFly
   // it doesn't need to implement void fly()
}


```

Interface vs Abstract class
Interfaces are simply Java's way of getting around the "Diamond problem" - which arrises due to multiple inheritance

Python - simply supports multiple inheritance, so you don't need interfaces - just use abstract classes

```py
from abc import ABC, abstractmethod

class Bird(ABC):
    hasBeak: bool
    wingSpan: int

    @abstractmethod
    def peck(self):
        raise NotImplementedError()

class ICanFly(ABC):
    @abstractmethod
    def fly(self):
        raise NotImplementedError()


class Sparrow(Bird, ICanFly):
    def fly(self):
        print("flap wings")


class Kiwi(Bird):
    ...
```

But to do this, didn't I modify existing code? Yes I did - and I want to avoid that.

When you design a system, you should design the system well - in advance

Staff Engineer / Principle Engineer / Senior Architect (10+ years of experience) In tier-1 companies (Google / Amazon / ...) - salary range - in India (Bangalore / Hyderabad)

upto 3 Cr (base package)

Because when you're a senior dev, you're expected to anticipate (predict) future requirement changes, and design code TODAY that addresses those future changes

YAGNI - you ain't gonna need it
KISS - keep it simple, stupid

both these apply in small teams / startups
these DO NOT apply in large companies or in projects with 10s or 100s of devs

---

✈ What else can fly?

```java
abstract class Bird {
   Boolean hasBeak;
   Integer wingSpan;
}

interface ICanFly {
   void fly();

   // flying process
   // make a small jump
   // spread wings
   // flap wings

   void flapWings();
   void kickToTakeOff();
}


class Sparrow extends Bird implements ICanFly {
   void fly() { print("fly low") }
}
class Pigeon extends Bird implements ICanFly {
   void fly() { print("fly overhead and poop bomb people") }
}
class Eagle extends Bird implements ICanFly {
   void fly() { print("glide elegantly high above") }
}

class Shaktiman implements ICanFly {
   void fly() {
      print("put up a finger, and sping round and round")
   }

   void flapWings() {
      // SORRY Shaktiman!
   }

}

```


Are there things apart from birds which can fly?

Aeroplanes, insects, bats, polythene bags, rockets, balloons, Shaktiman, Papa ki Pari, Mom's Chappal

>
> ❓ 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 - because Shaktiman can fly, but he doesn't have wings!

==========
⭐ **Interface Segregation Principle (ISP)**
==========

- Keep your interfaces minimal

- the client of your interface should not be forced to implement methods it doesn't need

How will you fix `ICanFly`?
Split it into 2 or more interfaces - ICanFly, IHasWings

```java
class Sparrow extends Bird implement ICanFly, IHasWings {
    ...
}

class Shaktiman implements ICanFly {
    ...
}
```

❓ Isn't this just the SRP applied to interfaces?
❓ Does this mean that SRP == ISP?

No - the intent is different

🔗 All SOLID principles are tightly linked to each other

We've done characters - Animals, Birds, Zoo Staff, Visitors ...

Let's now design some strcutres - Cages / Doors / ...

🟫 Design a Cage

1. High Level
   - abstractions (abstract classes / interfaces)
      - I declare some behavior, but I don't know how exactly this behavior will be executed
      - this is just a blueprint
      - it is abstract/incomplete
   - controllers (managerial code)

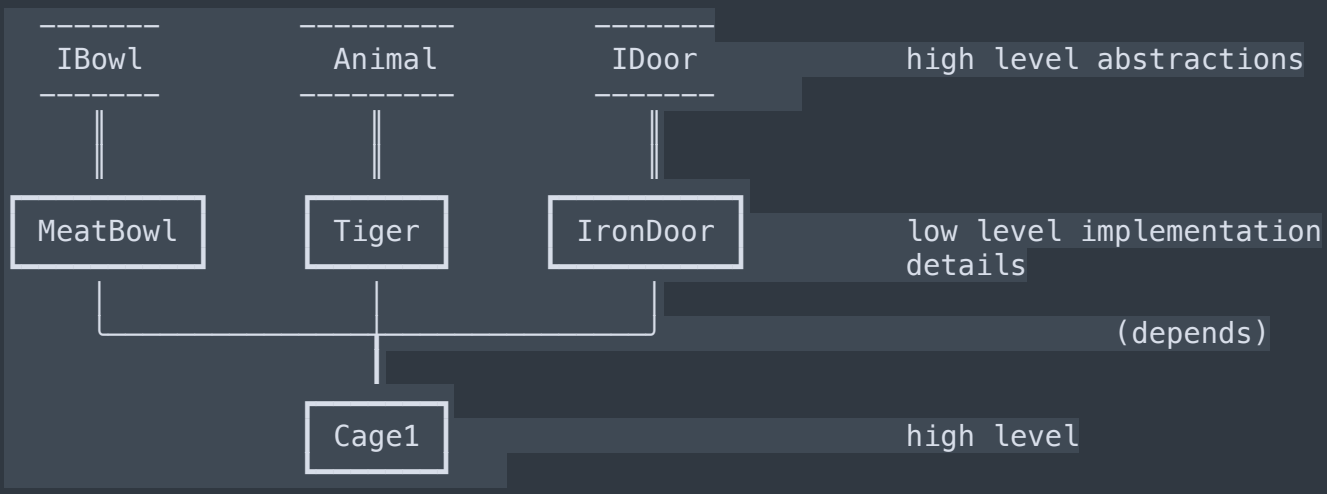
2. Low Level
   - implementation details (concerete classes)
      - we get to see exactly how something is being executed

```java

abstract class Animal {                 // high level abstraction
   String species;

   abstract void eat();
   abstract void poop();
   abstract void attack();
}
```

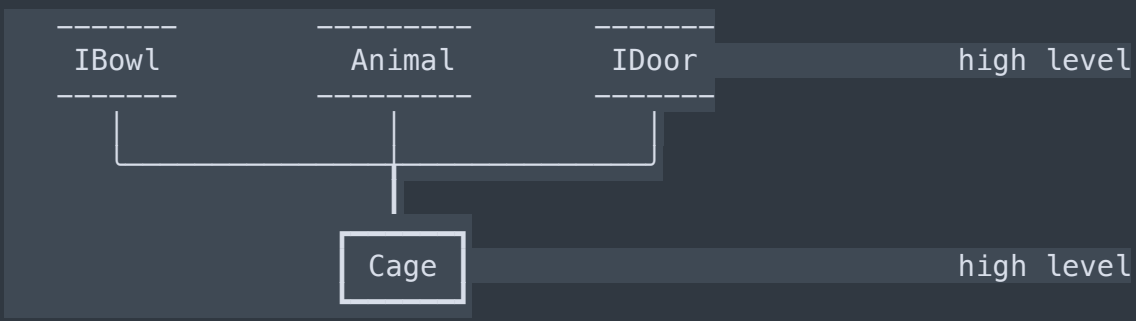
```
class Tiger extends Animal {            // low level implementation detail
   void eat() { print("eat the food, and the staff that brings the food") }
   void poop() { print("make smelly poop") }
   void attack() { print("rip off the throat") }
}

class Lion extends Animal { ... }       // low level implementation detail
class Human extends Animal { ... }      // low level implementation detail
class Monkey extends Animal { ... }     // low level implementation detail
class Wolverine extends Animal { ... } // low level implementation detail


interface IDoor {                       // high level abstraction
   void resistAttack(Attack attack);
   // declaration
}
class IronDoor implements IDoor {      // low level implementation detail
   // definition / implementation
   void resistAttack(Attack attack) {
      if(attack.stength < 100)
         print("successfully resisted")
      else
         print("door broken!")
   }
}
class WoodenDoor implement IDoor { ... } // low level implementation detail
class AdamantiumDoor implement IDoor { ... } // low level implementation detail


interface IBowl {                      // high level abstraction
   void feed(Animal animal);
}
class GrainBowl implement IBowl { ... } // low level implementation detail
class MeatBowl implement IBowl { ... }  // low level implementation detail
class FruitBowl implement IBowl { ... } // low level implementation detail

class Cage1 {                               // High Level controller
   // this is a cage for tigers
   MeatBowl bowl = new MeatBowl();
   IronDoor door = new IronDoor();
   List<Tiger> kitties = new ArrayList<>();

   public Cage1() {
      this.kitties.add(new Tiger(...), new Tiger(...), ...);
   }

   void resistAttack(Attack attack) {
      // delegate the task to the dependencies
      this.door.resistAttack(attack);
   }

   void feed() {
      for(Tiger t: this.kitties)
         this.bowl.feed(t);
   }
}

class Cage2 {
   // this is a cage for pigeons

   GrainBowl bowl = new GrainBowl();
   WoodenDoor door = new WoodenDoor();
   List<Pigeon> poopies = new ArrayList<>();
```

```
    public Cage2() {
        this.poopies.add(new Pigeon(...), ...);
    }
}
```

```

🐞 Lot of code repetition & redundancy
- In a zoo, I might have 100 different cages, and for that, I will need 100 different cage classes
- everytime I want to design a new cage, I need to create a new Cage class

```




```

==========

⭐ **Dependency Inversion Principle** - **goal - what we wish to do**

==========

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

```




```

But how?

==========
```

Dependency Injection - how to achieve the goal

- Instead of creating the dependencies yourself, you should let your client "inject" the dependencies
    - client - any piece of code that depends on you

```java
abstract class Animal { ... }                   // high level abstraction

class Tiger extends Animal { ... }             // low level implementation detail

class Lion extends Animal { ... }        // low level implementation detail
class Human extends Animal { ... }       // low level implementation detail
class Monkey extends Animal { ... }      // low level implementation detail
class Wolverine extends Animal { ... }  // low level implementation detail


interface IDoor { ... }                         // high level abstraction
class IronDoor implements IDoor { ... }  // low level implementation detail
class WoodenDoor implement IDoor { ... } // low level implementation detail
class AdamantiumDoor implement IDoor { ... } // low level implementation detail


interface IBowl { ... }                           // high level abstraction
class GrainBowl implement IBowl { ... } // low level implementation detail
class MeatBowl implement IBowl { ... }  // low level implementation detail
class FruitBowl implement IBowl { ... } // low level implementation detail

class Cage {                                  // High Level controller
   // this is now a GENERIC cage
   IBowl bowl;
   IDoor door;
   List<Animal> inhabitants = new ArrayList<>();

   // Inject the dependencies via constructor
   //                          injecting dependencies
   //          vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
   public Cage(IBowl bowl, IDoor door, List<Animal> inhabitants) {
      this.bowl = bowl;
      this.door = door;
      this.inhabitants.addAll(inhabitants);
   }

   void resistAttack(Attack attack) {
      this.door.resistAttack(attack);
   }

   void feed() {
      for(Tiger t: this.kitties)
         this.bowl.feed(t);
   }
}


class ZooGame {
   void main() {
      Cage forTigers = new Cage(
         new MeatBowl(...),
         new IronDoor(...),
         Arrays.asList(new Tiger(..), ...)
```

```
        );

        Cage forPigeons = new Cage(
            new GrainBowl(...),
            new WoodenDoor(...),
            Arrays.asList(new Pigeons(..), ...)
        );


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

    }
}
```

Enterprise Code
==========
https://github.com/EnterpriseQualityCoding/FizzBuzzEnterpriseEdition
This is satire, not actual code

1, 2, Fizz, 4, Buzz, Fizz, 7, 8, Fizz, Buzz, 11, Fizz, 13, 14, FizzBuzz, 16, ...

Code in large scale companies like Google will be "over-engineered"
Because the things that Google is optimizing for are different

An average project might last 10 years
1000s of devs will come and go
on a single small project, it could be be that 100 diff devs have worked on it

If you join Google
1. either you're good at LLD
   - you won't even have to read 90% of the code!
   - just looking at the class name will tell you exactly what it does!
2. you're bad at LLD
   - you will not be able to survive
   - you will complex classes
   - weird long ass names
   - everyone will be so complicated, that you will want to quit!

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

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

High Level Design Overview - important topics

---

1. Case study of bookmarking website - delicious
   - horizontal scaling
   - vertical scaling
   - DNS
   - load balancer (intro)
   - sharding (intro)
2. Load Balancing (deep dive)
   - health checks
   - various routing algorithms
   - Consistent Hashing
   - stateful vs stateless servers
   - how to configure AWS ELB
3. Caching
   - Pros and Cons of Caching
   - Invalidation strategies
      - write around
      - write back
      - write through
      - TTL
   - Eviction policies
   - Consistency of caches
4. Caching Case Studies
   - Scaler DSA code judge
   - Scaler contest leaderboard
   - Facebook news feed
5. SQL vs NoSQL
   - ACID
   - normalization
   - pros of SQL
   - cons of SQL - wrt sharding, distributed systems
   - Why NoSQL
   - various types of NoSQL databases
      - key value - Redis / DynamoDB
      - document dbs - MongoDB / Cockroach
      - Wide Column - Cassandra / Scylladb / Hbase
      - Blob/File storage - HDFS / S3 / Git LFS
      - Graph DBs - neo4j
      - vector DBS
      - timeseries
   - how to choose the database for a specific usecase
6. How to shard databases
   - how to choose a good sharding key
   - how to NOSQL database shard data
7. Internal of NOSQL
   - how to store data and indexes - LSM trees, Sparse Indexes, Bloom Filters
   - how to manage the shards - Orchestration, Zookeeper, Shard Creation, Data movement
8. CAP Theorem / PACELC theorem
   - master slave replication
      - various consistency levels
      - Quorum
      - Tunable consistency (R+W > X)
   - trade off b/w consistency, availability, & latency
   - when to choose what
9. Case Study - Google Typeahead
10. Case Study - Messaging apps (facebook messager, whatsapp, slack, discord)
11. Case Study - Messaging Queues (Kafka)
12. Case Study - Zookeeper
13. Case Study - Full text search (elastic search)

14. Case Study - Proximity based services - Uber
15. Popular small questions - Rate limiting, ID generation
16. Case Study - Large scale video pipelines (hotstar, youtube, twitch, scaler live class, google meet)
17. Microservices
    - event driven architecture
        - dead letter queues
    - pub/sub
    - consistency and distrubuted transactions
        - Saga
            - choreography
            - orchestration
        - compensating transactions
        - 2 phase commit
    - CQRS
    - Circuit breaker patter
    - monolith vs microservices
    - how to break a monolith into microservices
    - when not to use microservices
18. Mock interviews

🧩 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! ==========