

About the Instructor

=====

Pragy
Senior Software Engineer + Instructor @ Scaler

<https://linktr.ee/agarwal.pragy>


Agenda


=====


- ✓ 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)

 How long is this masterclass?
3 hours, with a 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) (<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

✅ Goals

=====

>
> ? 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?

- Learning and Research
 - reading other people's code
 - going through documentation
 - discussions
 - tutorials
 - chatGPT
 - stackoverflow
- Code Reviews (PRs)
- Debugging, Refactoring, Maintenance
- Collaboration and Meetings

- Scrum
 - Jira
 - Testing and QA
 - Documentation
 - DevOps and CI/CD
-
- Breaks: play TT, play snooker, chai, ..

How to maximize my off-time? By ensuring that whatever work I do, I do it right the first time!

We'd like to make our code

1. Readable
2. Testable
3. Maintainable
4. Extensible

=====

SOLID Principles

=====

- Single Responsibility Principle (SRP)
- Open/Closed Principle (OCP)
- Liskov's Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DSP)

O = Open Source / Open-Close

I = Isolation / Inversion of Control / Interface Segregation / Independence

D = Dependency Injection / Deployment / Dependency Inversion

Context

=====

Create a search engine like Google.com – can I complete this in a 3 hour masterclass?
No!

Curated example / toy problem – specifically designed to understand the concepts while minimizing complexity

- Game – Zoo 🐾
- Characters
 - Animals – birds, reptiles, humans, ..
 - Zoo Staff
 - Visitors
- Infrastructure
 - Cages
 - Feeding places
 - Roads

Programming language: pseudo-code

It will "look like" java, but it will not be java.

Any modern programming language that supports OOP

- Java (Kotlin / ..)
- Python
- C#
- C++
- Dart
- Ruby
- Javascript (Typescript / ..)
- Php

All these languages support SOLID

OOP: concepts map to classes ?



Design a Character

=====

```
```java
```

```
class ZooEntity {
 // attributes / properties / data
 // Staff
 String name;
 Gender gender;
 Integer age;
 Double salary;
 String designation;
 String department;
 // ...

 // Animal
 String name;
 Gender gender;
 Integer age;
 Boolean eatsMeat;
 Boolean canFly;
 Boolean hasWings;
 String species;

 // Visitor
 String name;
 Gender gender;
 Ticket ticket;
 Boolean isVIP;
 DateTime timeOfArrival;

 // methods / behaviour
 // Staff
 void sleep();
 void cleanPremises();
 void eat();
 void poop();
 void feedAnimals();
```

```

// Animal
void sleep();
void eat();
void poop();
void fly();
void fight();
void eatTheVisitor();

// Visitor
void roamAround();
void eat();
void petAnimals();
void getEatenByAnimals();
void litter();
}

class ZooEntityTester {
 void testAnimalEats() {
 ZooEntity animal = new ZooEntity(...);
 animal.eat(); // assert that something happens
 }
}

...

```

Minor Issue: Variable name conflicts

- name for staff and also for visitor
- easy to solve - we can just rename the variables
  - staffName
  - visitorName

🤖 Problems with the above code?

? Readable  
 Seems like it - I can read & understand this code

Issues:  
 - lengthy

- too much going on - it's a jumbled mess!
- understanding the code is difficult because I've to keep track of so many variables

? Testable

I can write testcases for it

Issues:

- changing the behaviour of animal can change the behavior of staff/visitor by mistake - because all the variables are shared
- this makes testing difficult!
  - code is "tightly coupled"

? Extensible

will come back to this later

? Maintainable

dev 1 - staff  
dev 2 - visitors  
dev 3 - animals

if all 3 devs make changes and push their code - merge conflicts!

if a new dev has to onboarded - they must be taught everything about animal+staff+visitor even if they want to work only on animals

🤔 What is the main reason for all these issues?

This code has too many responsibilities!

🔧 How to fix this?

=====  
★ **Single Responsibility Principle**  
=====

- Any function/class/module/package (unit-of-code) should have only 1, well-defined responsibility
  - only 1: not more
  - well defined: should not be vague
- if some piece of code doesn't follow the SRP, then we should split it into multiple parts.

OOP – provides a way of decomposing complex things: inheritance

```
```java
```

```
class ZooCharacter {  
    String name;  
    Integer age;  
    Gender gender;
```

```
    void eat();  
    void poop();  
    void sleep();  
}
```

```
class Staff extends ZooCharacter {  
    String designation;  
    Double salary;
```

```
    void cleanPremises();  
}
```

```
class Visitor extends ZooCharacter {  
    String ticketID;
```

```
    void roamAround();  
}
```

```
class Animal extends ZooCharacter {  
    Boolean canFly;  
    String species;
```



```
void eatTheVisitor();  
}  
```
```

Did we improve on any of the metrics?

Did we solve any of the issues?

Did we introduce any new issues?

## ? Readable

- New issue introduced: we've too many classes/files now
  - this is a non-issue!
  - you never work with all features at the same time
  - at any given time you're working on a specific (or a handful) of features
  - so any dev will only work with a handful of files
  - and each individual file is super tiny and easy to understand!

Yes, we improved the readability!

## ? Testable

If I make a change to class Animal, can that effect (by mistake) the working of the Staff class?

No!

Code has been de-coupled

Code is easier to test! I can test each component separately!

## ? Extensible

will come back to this later

## ? Maintainable

merge conflicts: significantly reduced!

onboard a new dev: easier to onboard

So is the code perfect now?

No! Far from it

Did we take a step in the right direction?

Yes we did!

## 🐦 Design a Bird

=====

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

    void fly() {

        !! what should we do here?

    }
}
```
```

🐦 different birds fly differently!

```
```java
class Bird extends Animal {
    String species; // inherited

    void fly() {
        if (species == "sparrow")
            print("flap wings vigorously and make noise")
        else if (species == "pigeon")
            print("fly above your head and poop on you")
        else if (species == "eagle")
            print("glide elegantly high up in the clouds")
        }
    }
}
```
```

Bad because it has an `if-else ladder`

🐞 Problems with the above code?

- Readable
- Testable
- Maintainable
  
- Extensible – FOCUS!

? Do we always write all code ourselves from scratch?

No! We use external libraries

```
```java
```

```
[PublicZooLibrary] {  
    // some author has coded this library and put it on github
```

```
    class Animal {  
        String species;  
    }
```

```
    class Bird extends Animal {  
        void fly() {  
            if (species == "sparrow")  
                print("flap wings vigorously and make noise")  
            else if (species == "pigeon")  
                print("fly above your head and poop on you")  
            else if (species == "eagle")  
                print("glide elegantly high up in the clouds")  
        }  
    }  
}
```

```
[MyCustomGame] {  
    import PublicZooLibrary.Bird;
```

```
    class AwesomeZooGame {  
        void main() {  
            Bird b = new Bird("pigeon");
```

```
        b.fly();
    }
}
```

? How can the "client" add a new Bird type?

```
```java
else if (species == "new bird type here")
 print("custom fly logic here ... ")
```
```

? Do we always have modification access?

No! I might not have modification access to this library!

- We might not have the source code – shipped as compiled files (.dll .exe .com .so .pyc .jar .war)
- even if we have the source, we might not have modification access

? How can we add a new type of Bird?

In this case it will be difficult. This code is NOT extensible!

🔧 How to fix this?

=====

★ Open/Close Principle

=====

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

? Why should code be "closed" for modification

? Why is it bad to modify existing code?

- Developer
 - write code on their laptop

- test it locally
- commit & push & generate a Pull Request (PR)
- PR will go for review
 - other devs in the team will suggest improvements
 - the dev will go and make changes to make those improvements
 - they will re-raise the PR
 - re-review
 - (iterative process)
 - PR gets merged
- Quality Assurance
 - write extra unit tests
 - write integration tests
 - UI: manual testing
 - update the docs
- Deployment Pipeline
 - + Staging servers
 - will ensure that the code doesn't break
 - there will be a lot of testing (unit/integration) & stress-testing
 - + Production servers
 - * A/B test
 - deploy to only 5% of the users
 - we will monitor a lot of metrics
 - number of exception
 - customer satisfaction
 - number of purchases
 - ...
 - * Deploy to 100% of the userbase

How much time does all this take?

It can take months for this entire process to complete for a new feature!

```
```java
```

```
[PublicZooLibrary] {
 // some author has coded this library and put it on github
```

```
 class Animal {
 String species;
```

```
}

abstract class Bird extends Animal {
 Integer beakLength; // because we've data, we are using an ABC instead of an interface

 abstract void fly();
}
```

```
class Sparrow extends Bird {
 void fly() {
 print("flap wings vigorously and make noise")
 }
}
```

```
class Pigeon extends Bird {
 void fly() {
 print("fly above your head and poop on you")
 }
}

class Eagle extends Bird {
 void fly() {
 print("glide elegantly high up in the clouds")
 }
}
}
```

```
[MyCustomGame] {
 import PublicZooLibrary.Bird;
```

```
 class Peacock extends Bird {
 void fly() {
 print("Males can't fly, but females (pe-hens) can")
 }
 }
}
```

```
class AwesomeZooGame {
 void main() {
 Peacock p = new Peacock();
 p.fly();
 }
}
```

```
}
```

```
...
```

- Was I able to add a new Bird type?  
Yes, the client was able to add it without having to modify the library code!
- Did I have to change existing code?
  - Didn't we modify the original Bird class?  
It seems that we did. But infact we did NOT!

OCP tells you that you will be able to add new features without modifying existing code, if you design for extensibility from the start!

? Isn't this the same thing that we did for Single Responsibility as well?

SRP - large ZooEntity class => broke it down into smaller subclasses  
OCP - large Bird class => broke it down into smaller subclasses

Yes, we did the exact same thing to implement the solution!

? Does that mean that OCP == SRP?

No!  
Solution was similar, but the intent was different

🔗 All the SOLID principles are tightly linked together!

Uncle Bob - Robert C. Martin

---

## ## Salary

- position: Staff Engineer / Principle Architect (senior - 10+ yoe)
- company: Google / Amazon (tier-1)
- location: Bengaluru/Hyderabad/Pune/Chennai (India)

Upto 3 Cr. in India!

Why would a company pay this much to 1 developer?

A good developer can anticipate future requirements and changes, and write code today that doesn't need to be modified for those new features!

Any design is bad if it needs to change in the face of new features

## Scaler LLD Cucciculum – 2 months

=====

Topics to learn to master Low Level Design

- Object Oriented Programming
  - inheritance
  - abstraction
  - encapsulation
  - generalization
  - polymorphism (runtime vs compile time)
  - multiple vs multi-level inheritance
  - Diamond problem & MRO
  - composition over inheritance
  - interfaces
- SOLID Principles
- Design Patterns
  - Structural / Behavioral / Creational
  - builder / singleton / factory / strategy / adapter / proxy / ...
- Database Schema design
  - (oop `impedance mismatch` with relational databases)
- Entities & Relationships (ER Diagram / Class Diagram)
- Case Studies



- Tic Tac Toe
  - Chess
  - Snake Ladder
  - Parking Lot
  - Splitwise
  - Library Management
- Testing & Test Driven Development
- REST API
- idempotency
  - naming conventions

### ### How do you know if you're learning the right thing?

Builder Pattern – how to implement this in Python?  
8% that said yes are WRONG!

Builder pattern

- works around language limitations
  - java does not have the following features
    - named arguments
    - change the position of arguments
    - validation for arguments
    - default values for arguments
  - you use builder pattern in java to work around these issues
- python/JS/C++/Kotlin: has all these features out of the box
- it is WRONG to implement builder pattern in any of these languages

### ### Solution

Always have a mentor who know what the correct thing is and can guide you on the correct thing

Where to learn this?

1. Free Masterclasses with certifications: <https://www.scaler.com/events>
2. Free courses with certifications: <https://www.scaler.com/topics/>
3. Comprehensive Program with Masters degree: <https://www.scaler.com/academy/>

---

Quick 10 mins break

---



Can all birds fly?

=====

```
```java
```

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

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

```
class Pigeon extends Bird {  
    void fly() {  
        print("fly on top of people and poop on their heads")  
    }  
}
```

```
class Eagle extends Bird {  
    void fly() {  
        print("glide elegantly high above")  
    }  
}
```

```
class Kiwi extends Bird {  
    void fly() {  
        !! what to do here?  
    }  
}
```

```
Not all birds can fly!  
Penguins, Ostrich, Kiwi, 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 – simply don't implement the `void fly()`

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

class Kiwi extends Bird {
 void eat() { ... }
 void poop() { ... }

 // void fly() will not be implemented here
}
```
```

🐛 Compiler Error! Either `class Kiwi extends Bird` must implement `void fly()` or it should itself be marked as abstract

The "abstract" keyword

- "abstract" means incomplete (in this context)
- `abstract class Bird` tells the compiler that Bird is an "incomplete" concept

⚠️ Throw an exception

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

class Kiwi extends Bird {
 void fly() {
 throw new FlightlessBirdException("Bro, I'm a Kiwi. Me no fly :'(")
 }
}
```
```

🐛 This violates expectations!

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

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

class Pigeon extends Bird {
 void fly() {
 print("fly on top of people and poop on their heads")
 }
}

class ZooGame {
 Bird getBirdObjectFromUserSelection() {

 // display a nice UI to the user
 // showcase various types of birds
 }
}
```

```
// user will select a type
// we will create an object of that type of bird
// we will return it
```

```
// if(userSelection == "pigeon")
// return new Pigeon(...)
// else if(userSelection == "sparrow")
// return new Sparrow(...)
```

```
}
```

```
void main() {
 Bird b = getBirdObjectFromUserSelection()
 b.fly()
}
```

```
}
```
```

Q: Can a variable of type Bird hold an object of type Sparrow?

Yes! Runtime Polymorphism. This is allowed because a sparrow is-a bird.

Anything I can do with the Bird variable, the sparrow will support that functionality



Before extension – Does this code work?

Works perfectly fine!

An intern comes to the company – and implements the class Kiwi

```
```java
class Kiwi extends Bird {
 void fly() {
 throw new FlightlessBirdException("...")
 }
}
```
```



After extension

– Did the intern modify your existing code?

No. They didn't touch it – they just "extended" the code

– Are you (original dev) aware of the fact that someone has added a Kiwi class?

Most likely no!

– Was the original code working before this addition was made?

Yes it was

– Is the code working now?

No! It is not!

Why?

```
```java
```

```
class ZooGame {
 Bird getBirdObjectFromUserSelection() {

 // earlier we had 2 birds – sparrow / pigeons
 // so this function could return any of them

 // but now we also have Kiwi
 // so this function can also return Kiwi

 }

 void main() {
 Bird b = getBirdObjectFromUserSelection()
 b.fly() // if b is a Kiwi, I will get an exception here!
 }
}
```

– Is the intern's code failing?

No.

– Whose code is failing?

Original code failed! Black magic – voodoo!

## ★ Liskov's Substitution Principle

- mathematical/type-theory definition: Any object of `class Parent` should be replaceable with any object of `class Child extends Parent` without causing any issues
- Indian intuition:
  - parents have expectations from their children (doctor/engineer)
  - child violates (cricketer)
  - parents set expectations, and the children are supposed to satisfy those expectations.

LSP: Children should not violate the expectations set by the parents!

Actual solution: parents should have reasonable expectations!

🎨 Redesign the system!

Don't have unreasonable expectations in the parent class

```
```java
```

```
abstract class Bird extends Animal {  
    abstract void eat() // It is guaranteed that every bird can eat  
  
    // abstract void fly() // this should not be here – because not all birds can fly!  
}
```

```
interface ICanFly {  
    void fly();  
}
```

```
class Sparrow extends Bird implements ICanFly {  
    void eat() { ... }  
  
    void fly() {  
        print(" ... ")  
    }  
}
```

```
}
```

```
class Kiwi extends Bird {  
    void eat() { ... }  
}
```

```
    // no need to implement void fly() here – no compiler error if you omit it  
}
```

```
...
```

```
```py  
from abc import ABC, abstractmethod
```

```
class Animal: ...
```

```
class Bird(ABC):
 @abstractmethod
 def eat(self): ...
```

```
class CanFlyMixin(ABC):
 @abstractmethod
 def fly(self): ...
```

```
class Sparrow(Bird, CanFlyMixin):
 def eat(self):
 print('...')
```

```
 def fly(self):
 print('...')
```

```
class Kiwi(Bird):
 def eat(self):
 print('...')
 # no need to implement void fly() here – no compiler error if you omit it
```

```
...
```



```

```java
// THIS IS A BAD SOLUTION!!
// Because it leads to Combinatorial explosion
// Imagine that instead of just Fly vs NoFly
// I have more such separations – Pecking vs NonPecking
//                               Running vs NonRunning
// I will have to implement all 2^N combinations
// PeckingNonFlyingBird
// NonPeckingNonFlyingRunningBird
// we prefer composition over inheritance

```

```

abstract class Bird {
}

```

```

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

```

```

abstract class NonFlyingBird extends Bird {
}
```

```

Q: didn't we modify existing code for this change to happen?

It's not modification – LSP says that you should think of this from the day-1

Q: aren't we violating the OCP?

No we're not – we're saying that the design should follow the SOLID principles from day-1

How will the code & main method look now?

```

```java

```

```

abstract class Bird extends Animal {
    abstract void eat() // It is guaranteed that every bird can eat

```

```

    // abstract void fly() // this should not be here – because not all birds can fly!
}

```

```
interface ICanFly {  
    void fly();  
}
```

```
class Sparrow extends Bird implements ICanFly {  
    void eat() { ... }  
  
    void fly() {  
        print(" ... ")  
    }  
}
```

```
class Kiwi extends Bird {  
    void eat() { ... }
```

```
    // no need to implement void fly() here – no compiler error if you omit it  
}
```

```
class ZooGame {  
    Bird getBirdObjectFromUserSelection() {  
  
        // earlier we had 2 birds – sparrow / pigeons  
        // so this function could return any of them  
  
        // but now we also have Kiwi  
        // so this function can also return Kiwi  
  
    }  
}
```

```
void main() {  
    Bird b = getBirdObjectFromUserSelection()  
    // b.fly()      // compiler error! fly() does not exist in class Bird!  
  
    // explicit downcasting  
    if(b instanceof ICanFly) {  
        ICanFly flyingBird = (ICanFly) b;  
        flyingBird.fly();  
    }  
}
```

```
}
```

```
...
```



What else can fly?

=====

What steps does a Bird take to fly?

- jump
- spread wings
- flap wings
- fly

```
```java
```

```
abstract class Bird {
 ...
}
```

```
interface ICanFly {
 void fly();

 void flapWings();
 void spreadWings();
 void jump();
}
```

```
class Sparrow extends Bird implements ICanFly {
 void fly()
}
```

```
class Kiwi extends Bird {
```

```
// there's no need to implement void fly()
}

class Shaktiman implements ICanFly {
 void fly() {
 print("Raise finger, spin fast, make weird noises")
 }

 void flapWings() {
 // SORRY Shaktiman!
 }
}

...
```

What else can fly (apart from birds)?

Kite (patang), aeroplanes, rockets, drones, helicopters, insects, balloons, Shaktiman, Mom's chappal, Hanuman Ji

```
>
> ? 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]
>
```

## ===== ★ Interface Segregation Principle =====

- Keep your interfaces minimal
- A client/user of your interface should not be forced to implement a method that it doesn't need / doesn't support

```
```java
```

```
interface DatabaseCursor {  
    List<Row> find(String query)  
    List<Row> insert(String query)  
    List<Row> delete(String query)  
    List<Row> join(String query, String table1, String table2)  
}
```

```
class PostgreSQLDatabaseCursor implements DatabaseCursor {  
    List<Row> find(String query)  
    List<Row> insert(String query)  
    List<Row> delete(String query)  
  
    List<Row> join(String query, String table1, String table2)  
}
```

```
class SQLiteDatabaseCursor implements DatabaseCursor {  
    List<Row> find(String query)  
    List<Row> insert(String query)  
    List<Row> delete(String query)  
  
    List<Row> join(String query, String table1, String table2)  
}
```

```
class OracleDBDatabaseCursor implements DatabaseCursor {  
    List<Row> find(String query)  
    List<Row> insert(String query)  
    List<Row> delete(String query)  
  
    List<Row> join(String query, String table1, String table2)  
}
```

```
class MongoDBDatabaseCursor implements DatabaseCursor {  
    List<Row> find(String query)  
    List<Row> insert(String query)  
    List<Row> delete(String query)
```

```
// but MongoDB doesn't support joins!
}

...

```

? Isn't this similar to LSP? Isn't this just SRP applied to interfaces?
Yes. The solution is same – but intents are different

🔗 All the SOLID principles are tightly linked

How will you fix `ICanFly`?

```
```java

```

```
interface ICanFly {
 void fly();
}
```

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

```
interface ICanJump {
 void jump();
}
```

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

```
class Shaktiman implements ICanFly {
 void fly() { ... }
}
```

```
}
...
```

## Design a Cage

=====

```
```java
```

```
interface IBowl {                                // High Level Abstraction  
    void feed(Animal animal);  
}  
class MeatBowl implements IBowl {                // Low Level Implementation Detail  
    void feed(Animal animal) {  
        // measure 2 ounces of meat  
        // make sure there are no bones so animals don't choke  
        // chop into bite sized pieces  
        // add enzymes for digestion  
        // add flavoring  
        // make sure the food is at the right temperature  
        // serve  
    }  
}  
class FruitBowl implements IBowl {               // Low Level Implementation Detail  
    void feed(Animal animal) {  
        ...  
    }  
}  
  
interface IDoor {                                // High Level Abstraction  
    void resistAttack(Attack attack);  
}  
class IronDoor implements IDoor {                // Low Level Implementation Detail  
    void resistAttack(Attack attack) {  
        if (attack.power < IRON_DOOR_STRENGTH)  
            print("attack mitigated")  
        else  
            print("door broken - all animals have escaped. All visitors got eaten")  
    }  
}
```

```
}  
class WoodenDoor implements IDoor { // Low Level Implementation Detail  
    void resistAttack(Attack attack) {  
        ...  
    }  
}
```

```
class Cage1 {
```

```
    // this cage will be for tigers  
    MeatBowl bowl = new MeatBowl(); // instantiated my dependencies  
    IronDoor door = new IronDoor(); // ...  
    List<Tiger> kitties = new ArrayList();
```

```
    public Cage1() {  
        this.kitties.add(new Tiger("simba"));  
        this.kitties.add(new Tiger("musafa"));  
        this.kitties.add(new Tiger("scar"));  
    }
```

```
    void resistAttack(Attack attack) {  
        this.door.resistAttack(attack); // delegate the responsibility  
                                         // to my dependency  
    }
```

```
    void feedAnimals() {  
        for(Tiger kitty: this.kitties)  
            this.bowl.feed(kitty) // delegate  
    }  
}
```

```
class Cage2 {
```

```
    // this cage will be for peacocks  
    FruitBowl bowl = new FruitBowl(); // instantiated my dependencies  
    WoodenDoor door = new WoodenDoor(); // ...  
    List<Peacock> peas = new ArrayList();
```



```

public Cage2() {
    this.peas.add(new Peacock("pea1"));
    this.peas.add(new Peacock("pea2"));
    this.peas.add(new Peacock("pea3"));
}

void resistAttack(Attack attack) {
    this.door.resistAttack(attack); // delegate the responsibility
                                    // to my dependency
}

void feedAnimals() {
    for(Tiger kitty: this.peas)
        this.bowl.feed(kitty) // delegate
}
}

class ZooGame {
    void main() {
        Cage1 tigerCage = new Cage1()
        tigerCage.resistAttack(new Attack(...))
    }
}
```

```



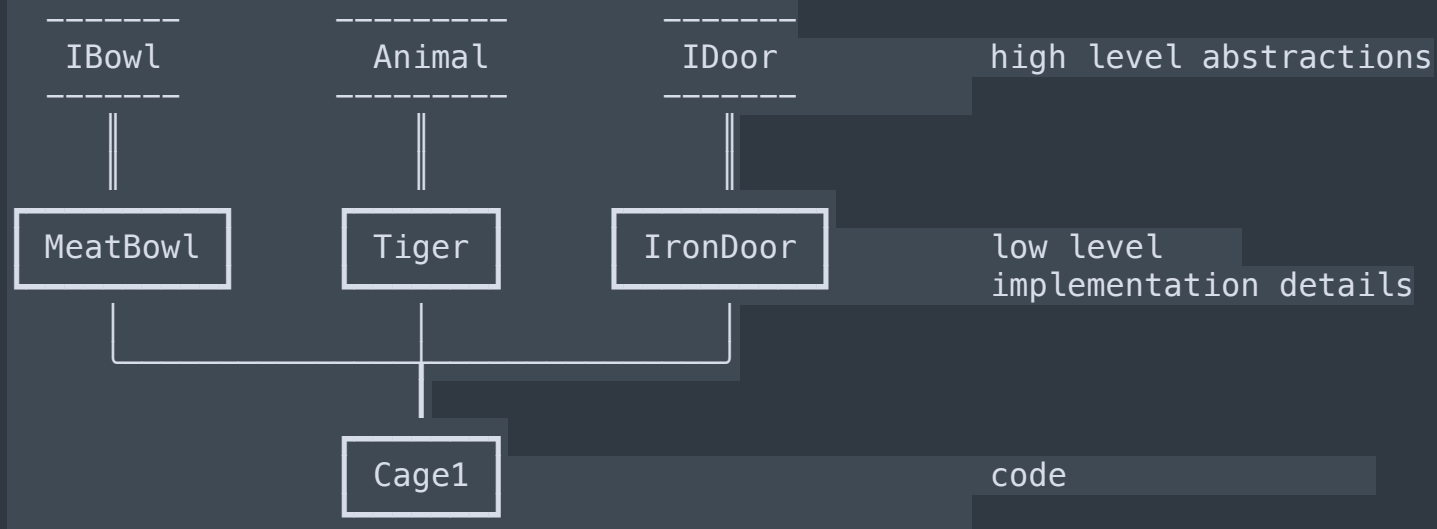
Violates DRY (don't repeat yourself)

#### #### High Level vs Low Level code

- High Level abstraction: tells us what to do, but not how to do it
- Low Level Implementation Details: tells you exactly how to do something (step by step)

...

interface      abstract class      interface



...

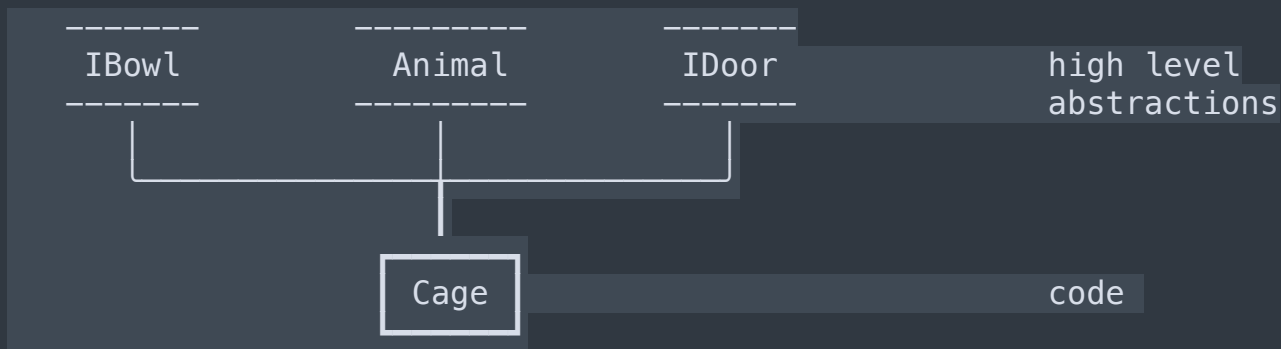
`class Cage1` depends on low level implementation details `MeatBowl`, `Tiger`, and `IronDoor`

## ★ Dependency Inversion Principle

what to do

- Code should NOT depend on low level implementation details
- Code should ONLY depend on high level abstractions

...



...

But how?

## =====

### 🔪 Dependency Injection

## =====

### how to achieve inversion

- do not instantiate your dependencies yourself
- instead, let your client (whoever is using you) "inject" the dependencies into you

```
```java
```

```
interface IDoor { ... } // High Level Abstraction
class IronDoor implements IDoor { ... } // Low Level Implementation Detail
class WoodenDoor implements IDoor { ... } // Low Level Implementation Detail
class AdamantiumDoor implements IDoor { ... } // Low Level Implementation Detail
```

```
interface IBowl { ... } // High Level Abstraction
class MeatBowl implements IBowl { ... } // Low Level Implementation Detail
class GrainBowl implements IBowl { ... } // Low Level Implementation Detail
class FruitBowl implements IBowl { ... } // Low Level Implementation Detail
```

```
class Cage {
    // generic cage

    IBowl bowl; // declare the dependencies
    IDoor door; // but don't instantiate them
    List<Animal> animals;
```

```
    // client will inject the dependency
    // via constructor (in this case - there are other ways of injection as well)
    public Cage(IBowl bowl, IDoor door, List<Animal> animals) {
        this.bowl = bowl
        this.door = door
        this.animals = animals
    }
```

```

    void resistAttack(Attack attack) {
        this.door.resistAttack(attack); // delegate the responsibility
                                        // to my dependency
    }

    void feedAnimals() {
        for(Animals animal: this.animals)
            this.bowl.feed(animal) // delegate
    }
}

class ZooGame {
    void main() {
        Cage tigerCage = new Cage(
            new MeatBowl(),
            new IronDoor(),
            Arrays.asList(new Tiger("simba"), new Tiger("musafa"))
        );

        Cage peacockCage = new Cage(
            new FruitBowl(),
            new WooderDoor(),
            Arrays.asList(new Peacock("pea1"), new Peacock("pea2"))
        );

        Cage xmenCage = new Cage(
            new MeatBowl(),
            new AdamantiumDoor(),
            Arrays.asList(new Wolverine(), new Deadpool())
        );

    }
}

```

- SpringBoot (java)
- Rails (ruby)
- Django (python)
- Laravel (php)

Frontend

- React (js)
- Flutter (Dart)
- Angular (js)

All these frameworks work using DI (dependency inversion+injection)

Inversion of Control: framework is in control – you're just providing the hooks/callback implementations.

Enterprise Code

When you go to companies like Google

- you will "over engineered" code
 - design patterns everywhere
 - very long names everywhere
 - `abstract class AbstractDatabaseQueryBuilderStrategy``
- even when it is not strictly needed

If you're a dev who is not good at LLD

- but you will have a very hard time
- you will not be able to understand the code
- be put into a PIP (Performance improvement proposal)

If you're dev who is good at LLD

- 90% of the time you won't even have to read the code
- because the class name will tell you exactly what the class does!

LLD: all about building a toolkit of powerful solutions to common problems

=====

🎁 Bonus Content

=====

🧩 Assignment

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

QnA

1. How can I be consistent in my coding journey?

GATE – 5 days a week – watch One Piece

- 2 days – go to a coaching

- 6 hours a day

- I had no other choice but to study

- structure / curriculum really helps with consistency

Solution: have someone else enforce a structure on you

2. How to implement SOLID principles in my project

Practice! Complete the assignment – ping me on linkedin for personal feedback

3. How important is SOLID principles for web development?

"[framework] inversion of control"

"[framework] design patterns"

"[framework] components"

4. Do I have to implement all SOLID principles in my code, or just some?

SOLID are guidelines – they're not rules.

Rules: enforced (don't murder – or the police will take you)

Guidelines: good to have (don't litter – no enforcement, but not littering will give you benefits)

later)

You do NOT have to implement all the SOLID principle
However if you implement just a few (SRP + DI) you might automatically get the others for free

5. When should we use Interfaces vs Abstract Classes

Interfaces don't have state (no variables) – they're just to enforce a contract/protocol/API

Abstract classes are basically interfaces with the additional ability to store data (variables)

By default, use interface

If you need state, change that interface to an abstract class

★ 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

>

>

> We all need people who will give us feedback.

> That's how we improve.  Bill Gates

>

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