Clean Code?

WHY ⇒ readability ↑, easy to debug, scalable & available, modular, object-oriented, reusability ↑, easy to maintain & work in collaboration, simple/structured code.

HOW ⇒ following a set of guidelines/rules known as design principles

↳ subjective in nature

→ variables, functions/methods, classes & objects, interfaces, packages having meaningful names.

- descriptive
- intent-revealing
- following naming conventions
  - camelcase
  - interface - 'I'
  - packages
    - subpackages
      - com.archit.ild.dsa
- Short & crisp
- distinctive naming
  - do not use integers in names
    - eg a0, a1, a2, ...

eg variables names
- ~~x~~, ~~y~~, ~~itsOnRow~~, ~~itsOnCol~~

eg function names ⇒ VERB
- getters & setters → getName, setName
- predefined → sort, reverse, swap, etc

eg Classes ⇒ NOUN
- eg Student, Movie, Theater, etc

eg Interfaces & Generalization
- eg ICar, IStudent, IUser, etc

eg Account
- SalaryAcc
- CurrentAcc
- SavingsAcc

Good functions/methods

1. function name → meaningful
2. Arguments or Parameter → 0, 1, 2
3. body → short & crisp
   - helper functions!
     - nCr
       - calcfact(n)
       - calcfact(n-r)
       - calcfact(r)
     - frequencySort
       - freq array
       - bucket array
       - result
   - ≤ 20 lines of code → screen w/o scrolling

2. addPayment(User debtor, User creditor, int payment, String date, String message)

vs

2. addPayment(Transaction transaction)

④ Single objective / responsibility

↳ Create / Read / Update / Delete
getter setter remove

↳ Separation of concerns !

⑤ modular → functions should be having codes of different modules.

→ Proper indented code with use of comments, annotations (meta-data), documentation, etc.

→ Instead of returning from method, throw Exceptions!

→ Do proper Error handling, unit testing, etc!

eg

```
public String remove(Student id){
    if(db.find(id)==false){
        return "sid not found";
    }
    db.remove(id);
    return "user deleted";
}
```

poor ☹

better ☺

```
public void remove(Student id){
    if(db.find(id)==false)
        throw new StNotFoundException();
    db.remove(id);
}
```

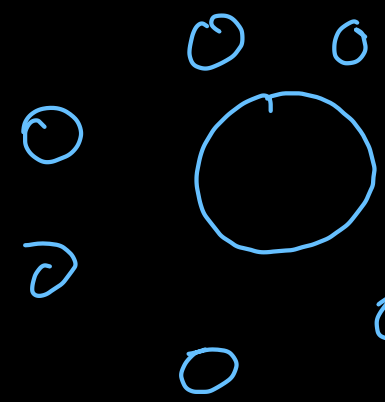
→ Modular programming

- functions / methods
- classes & objects
- microservices
- MVC architecture

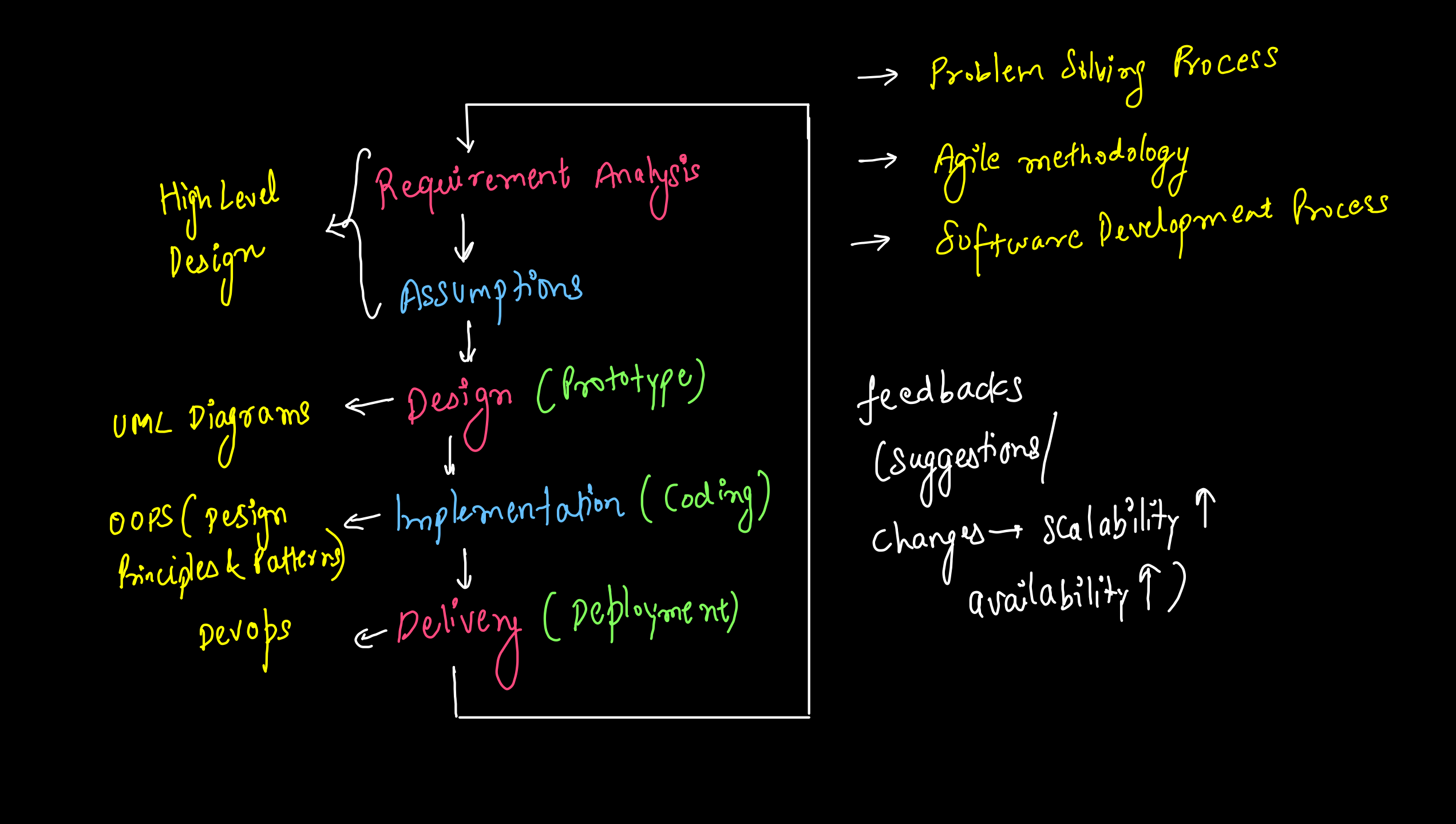
monolithic architecture

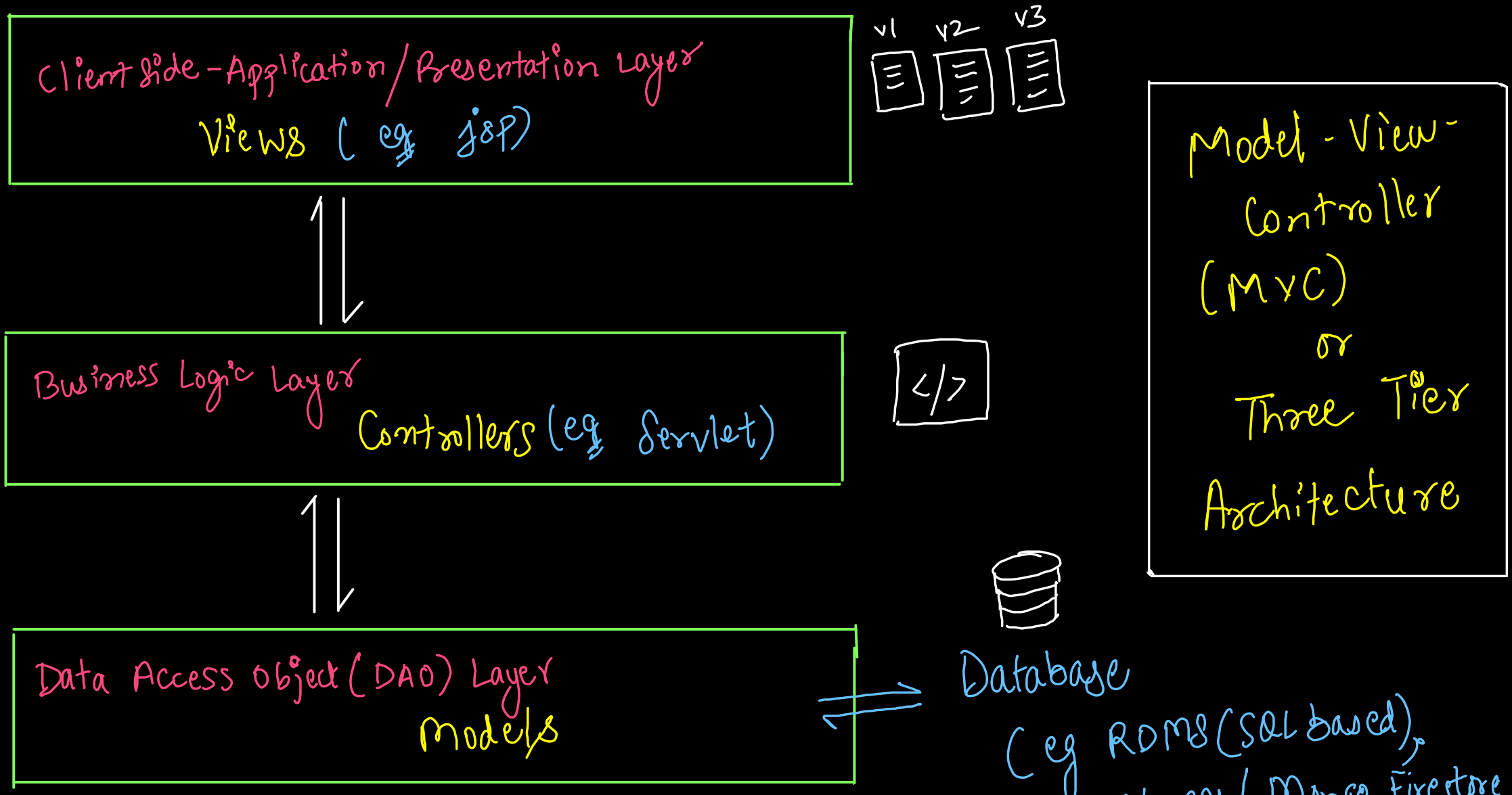
All functions in same service

microservice (master-slave)



- sharded database
- modules
- load balancer
- redis cache


→ Problem Solving Process
→ Agile methodology
→ Software Development Process
High Level Design
Requirement Analysis
Assumptions
Design (Prototype)
UML Diagrams
Implementation (Coding)
OOPS (Design Principles & Patterns)
Delivery (Deployment)
Devops
feedbacks (suggestions/ Changes → scalability ↑ availability ↑)


Client Side - Application / Presentation Layer
Views (eg jsp)
v1
v2
v3
Business Logic Layer
Controllers (eg Servlet)
Data Access Object (DAO) Layer
Models
Database
(eg RDMS (SQL based),
NonSQL (Mongo, Firestore
, etc)
Model - View - Controller
(MVC)
or
Three Tier
Architecture

Design Principles

⇒ Important Design Principles

- DRY (Don't Repeat Yourself)
- KISS (Keep It Simple Stupid)
- YAGNI (You Ain't Gonna Need It)
- CQS (Command - Query Separation)
- Maximum Cohesion & Minimum Coupling
- Composition (hasA) over Inheritance (isA)

SOLID Design Principles

- (S) : Single Responsibility Principle
- (O) : Open-Closed Principle
- (L) : Liskov Substitution Principle
- (I) : Interface Segregation Principle
- (D) : Dependency Inversion Principle

⇒ Other Design Principles

- Curly's law
- Boy Scout's law
- Murphy's law
- Avoid Premature Optimizatn
- Law of Demeter

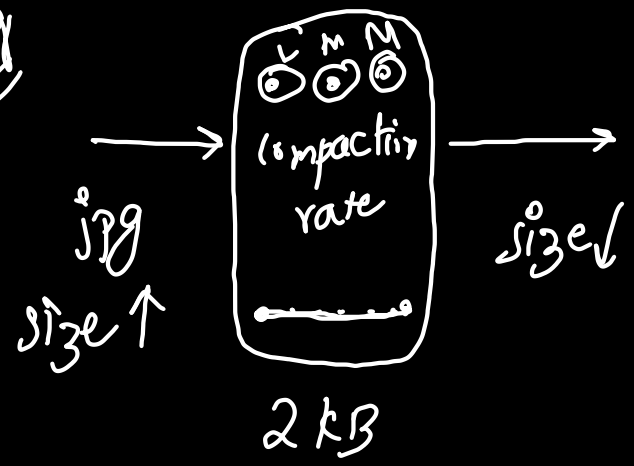
① KISS {Keep It Simple Stupid}

"Most systems work best if they are kept simple rather than made complex"

↳ over-engineering ②

Why ⇒ less time to write the code
⇒ readable ↑ ⇒ clean code

Eg



Most systems work best if they are kept simple rather than made complex.

Why

- Less code takes less time to write, has less bugs, and is easier to modify.
- Simplicity is the ultimate sophistication.
- It seems that perfection is reached not when there is nothing left to add, but when there is nothing left to take away.

A solution that follows the KISS principle might look boring or even “stupid” but simple and understandable. The KISS principle states that there is no value in a solution being “clever” but being easily understandable.

This does not mean that features like inheritance and polymorphism should not be used at all. Rather they should only be used when they are necessary or there is some substantial advantage.

Strategies

This is a very general principle, so there is a large variety of possible strategies to adhere more to this principle largely depending on the given design problem:

- Avoid inheritance, polymorphism, dynamic binding and other complicated OOP concepts. Use delegation and simple if-constructs instead.
- Avoid low-level optimization of algorithms, especially when involving Assembler, bit-operations, and pointers. Slower implementations will work just fine.
- Use simple brute-force solutions instead of complicated algorithms. Slower algorithms will work in the first place.
- Avoid numerous classes and methods as well as large code blocks (see More Is More Complex)
- For slightly unrelated but rather small pieces of functionality use private methods instead of an additional class.
- Avoid general solutions needing parameterization. A specific solution will suffice.

eg Division by 2 → by bit { n/2 n<<1 }

eg → split string sentence into sentences on space.

→ reduce code division into multiple classes.

↳ avoid generic programming / interface

Example

```java
public String weekday1(int dayOfWeek)
{
    switch (dayOfWeek)
    {
        case 1: return "Monday";
        case 2: return "Tuesday";
        case 3: return "Wednesday";
        case 4: return "Thursday";
        case 5: return "Friday";
        case 6: return "Saturday";
        case 7: return "Sunday";
        default: throw new IllegalArgumentException("dayOfWeek must be in range 1..7");
    }
}

public String weekday2(int dayOfWeek)
{
    if ((dayOfWeek < 1) || (dayOfWeek > 7))
        throw new IllegalArgumentException("dayOfWeek must be in range 1..7");

    final String[] weekdays = {
        "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"};

    return weekdays[dayOfWeek-1];
}
```

Prefer simple code

② YAGNI { You Aren't Gonna Need it }

↳ over Engineering

YAGNI stands for "you aren't gonna need it": don't implement something until it is necessary.

Why

- Any work that's only used for a feature that's needed tomorrow, means losing effort from features that need to be done for the current iteration.
- It leads to code bloat; the software becomes larger and more complicated.

How

- Always implement things when you actually need them, never when you just foresee that you need them.

Example (1)

You find that you need a getter for some instance variable. Fine, write it. Don't write the setter because "we're going to need it". Don't write getters for other instance variables because "we're going to need them".

The best way to implement code quickly is to implement less of it. The best way to have fewer bugs is to implement less code.

Example (2)

For example: when we will have to do validation of email and password field, we shouldn't do validation of name, because we may never need it.

③ DRY { Don't Repeat Yourself }

- code redundancy ↓
- Code duplicate ↓
- generic programming ↑

Definition

Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.

Each significant piece of functionality in a program should be implemented in just one place in the source code. Where similar functions are carried out by distinct pieces of code, it is generally beneficial to combine them into one by abstracting out the varying parts.

scalable ↓
new feature ↓
refactor ↓

Why

- Duplication (inadvertent or purposeful duplication) can lead to maintenance nightmares, poor factoring, and logical contradictions.
- A modification of any single element of a system does not require a change in other logically unrelated elements.
- Additionally, elements that are logically related all change predictably and uniformly, and are thus kept in sync.

How

→ controller / service

→ constants → enum

external dependencies → yaml, xml

- Put business rules, long expressions, if statements, math formulas, metadata, etc. in only one place.
- Identify the single, definitive source of every piece of knowledge used in your system, and then use that source to generate applicable instances of that knowledge (code, documentation, tests, etc).

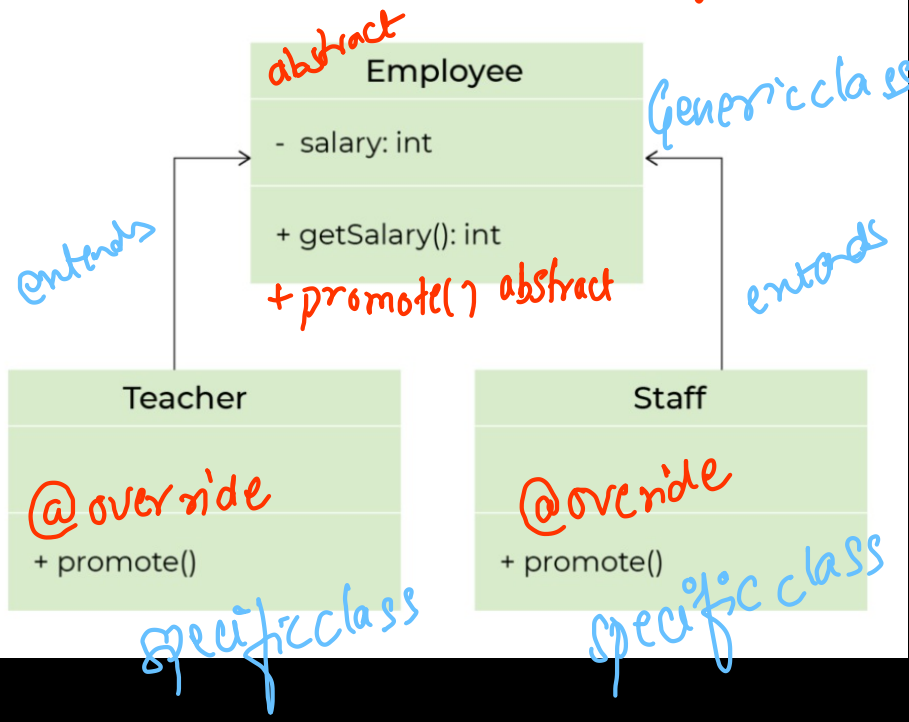
Example!

Naive implementation (code violating DRY)

Teacher
- salary: int
+ getSalary(): int + promote()

Staff
- salary: int
+ getSalary(): int + promote()

Better implementation (Generalization)



Example2 : Code violating DRY principle

```java
public class GFG {

    // Method 1
    // For cse department
    public void CSE()
    {
        System.out.println("This is computer science");
    }

    // Method 2
    // For cse dept. college
    public void college()
    {
        System.out.println("IIT - Madras");
    }
    // Method 3
    // ece dept method
    public void ECE()
    {
        System.out.println("This is electronics");
    }

    // Method 4
    // For ece dept college 1
    public void college1()
    {
        System.out.println("IIT - Madras");
    }
    // Method 5
    // For IT dept
    public void IT()
    {
        System.out.println(
            "This is Information Technology");
    }

    // Method 6
    // For IT dept college 2
    public void college2()
    {
        System.out.println("IIT - Madras");
    }

    // Method 7
    // Main driver method
    public static void main(String[] args)
    {

        // Creating object of class in main() method
        GFG s = new GFG();

        // Calling above methods one by one
        // as created above
        s.CSE();
        s.college();
        s.ECE();
        s.college1();
        s.IT();
        s.college2();
    }
}
```

redundant

redundant

Applying DRY principle

```java
public class GFG {

    // Method 1
    // For cse department
    public void CSE()
    {

        // Print statement
        System.out.println("This is computer science");

        // Calling method
        college();
    }

    // Method 2
    // For ece dept method
    public void ECE()
    {
        System.out.println("This is electronics");

        // Calling method
        college();
    }

    // Method 3
    // For IT dept
    public void IT()
    {

        // Print statement
        System.out.println(
            "This is Information Technology");

        // Callling method
        college();
    }
```

```java
    // Method 4
    // For college dept
    public void college()
    {

        // Print statement
        System.out.println("IIT - Madras");
    }

    // Method 5
    // Main driver method
    public static void main(String[] args)
    {

        // Creating object of class in main() method
        GFG s = new GFG();

        // Calling the methods one by one
        // as created above
        s.CSE();
        s.ECE();
        s.IT();
    }
```

(4) Avoid Premature Optimizations

"Pre optimization is the root of all evils."

Why

- It is unknown upfront where the bottlenecks will be.
- After optimization, it might be harder to read and thus maintain.

How

- Make It Work Make It Right Make It Fast
- Don't optimize until you need to, and only after profiling you discover a bottleneck optimise that.

Example 1

```
int i, sum = 0;
for (i = 1; i <= N; ++i) {
  sum += i;
}
printf("sum: %d\n", sum);
```

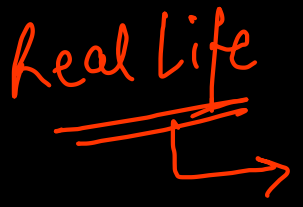
optimized →

```
int sum = N * (1 + N) / 2;
printf("sum: %d\n", sum);
```

Example 2 N<16 InsertionSort → MergeSort
better
overhead
recursion

Problems

1. integer overflow while multiplication
2. optimized version is not much intuitive
3. If N is small, '+' will perform faster/better than '*'



Examples of premature optimization

People engage in premature optimization in many ways and in many areas of life. This includes, for example:

- Spending a lot of resources (e.g., time and effort) trying to optimize certain functions in a codebase early on, even though these optimizations are likely to be irrelevant later, due to necessary changes in the code.
- Spending a lot of resources (e.g., time and effort) trying to structure a startup in a way that will allow it to scale to hundreds of millions of users, before having acquired even a single one.
- Spending a lot of resources (e.g., time and money) incorporating a company legally, before checking that anyone is interested in the product that it will sell.
- Spending a lot of resources (e.g., time and money) picking out the best possible gear for a hobby, before actually starting the hobby.

(5) Boy Scout's Law

↳ "we should always leave the code cleaner than we found it"

projects → workitems ↑ → code quality ↓
↳ backlog or technical debt

Why

backlog WI

- When making changes to an existing codebase the code quality tends to degrade, accumulating technical debt. Following the boyscout rule, we should mind the quality with each commit. Technical debt is resisted by continuous refactoring, no matter how small.

How

- With each commit make sure it does not degrade the codebase quality.
- Any time someone sees some code that isn't as clear as it should be, they should take the opportunity to fix it right there and then.

Examples

(backward compatibility)

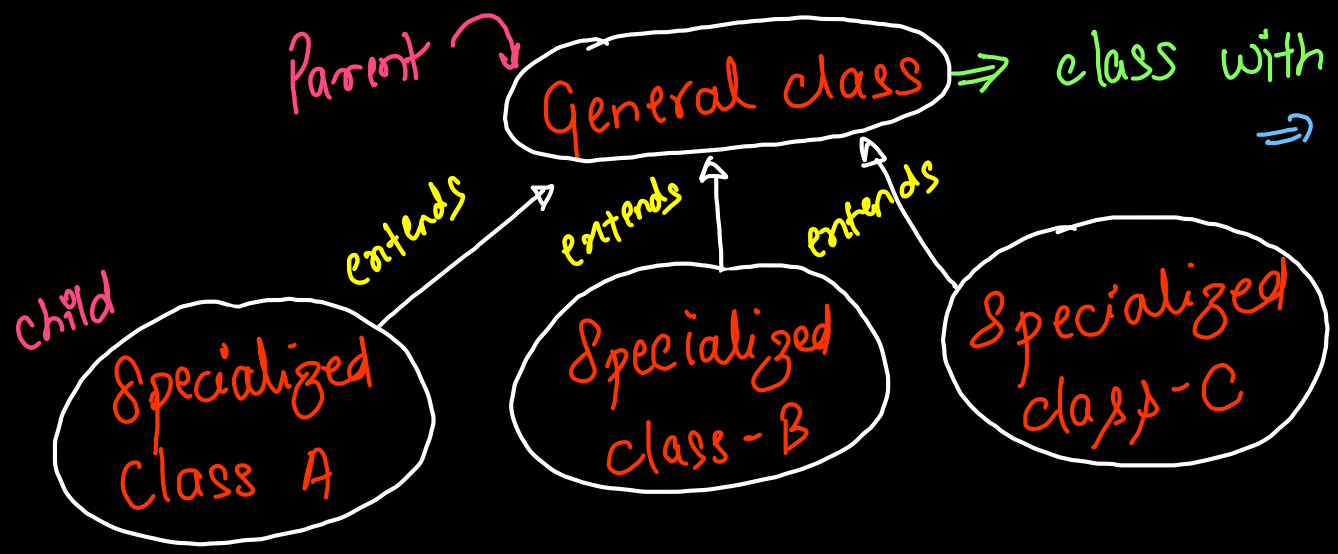
1. Deprecated legacy codes {@Deprecated Annotation}
2. unused codes removal
   - getters/setters
   - overloaded constructors
   - private methods
3. Code refactoring
   - Improve indentation
   - add comments/documentation
   - follow DRY principle.

⑥ Abstraction

- Generalization
  - generic classes
  - generic programming!
- Interfaces & abstract classes
  - for easy maintainance (adding new features becomes easy)

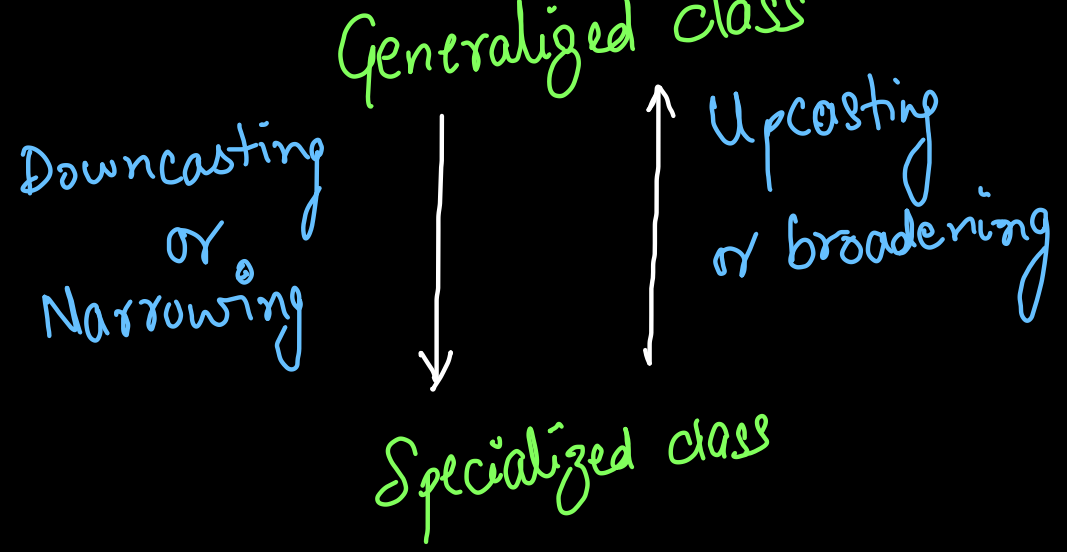
hide implementation details and does not leak any unnecessary information.

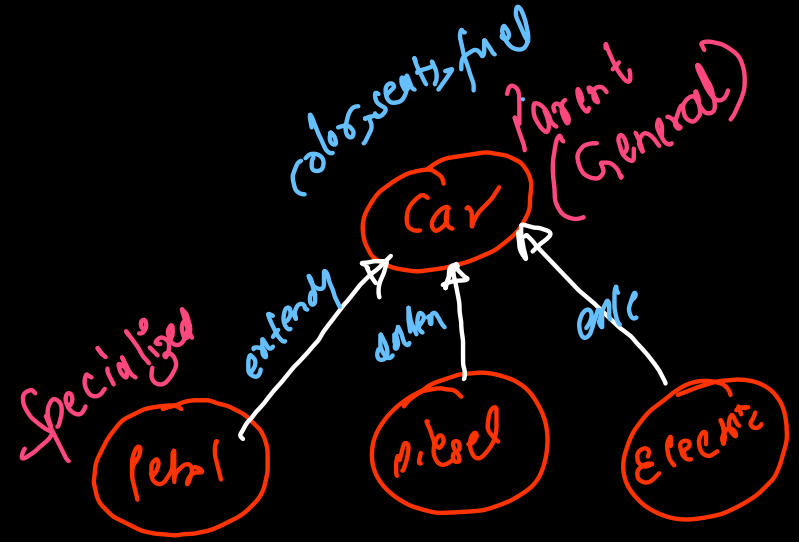
Generalization & Specialization



⇒ class with generic/main properties & functions
⇒ usually does not exist in real world.

⇒ classes which are specific, having particular states!
⇒ exist in real world





A software module hides information (i.e. implementation details) by providing an interface, and not leak any unnecessary information.

Why

- When the implementation changes, the interface clients are using does not have to change.

How

- Minimize accessibility of classes and members.
- Don’t expose member data in public.
- Avoid putting private implementation details into a class’s interface.
- Decrease coupling to hide more implementation details.

(7) Maximum Cohesion & Minimum Coupling

Cohesion → "Degree to which how strongly related are various responsibilities in a given module?"

Coupling → "Degree to which a given module relates or depends on other module?"

Cohesion ?

"Code that changes together stays together"

How? → Divide a class so that each module (class) takes a single responsibility.

→ Group related functionalities sharing a single responsibility in a class/package/microservice/etc.

A module with high cohesion contains elements that are tightly related to each other and united in their purpose. For example, all the methods within a *User* class should represent the user behavior.

A module is said to have low cohesion if it contains unrelated elements. For example, a *User* class containing a method on how to validate the email address. *User* class can be responsible for storing the email address of the user but not for validating it or sending an email:

Pictorial view of high cohesion and low cohesion:

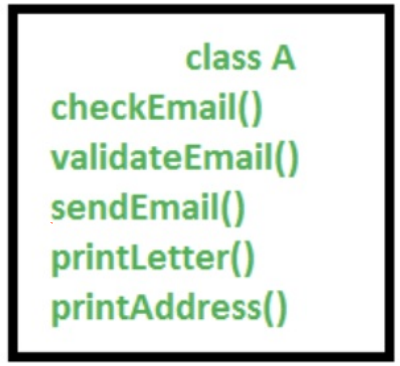


Fig: Low cohesion

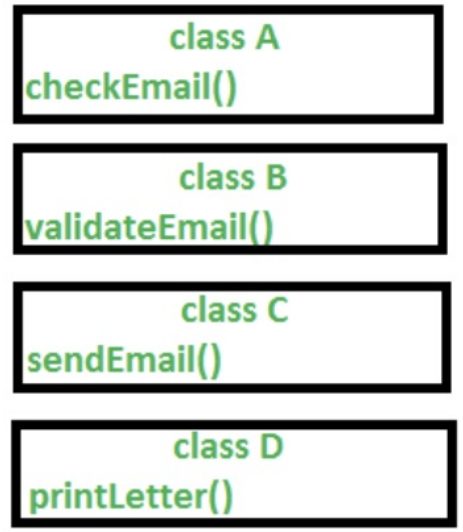


Fig: High cohesion

↷ closely related to Single Responsibility Principle (SRP).

⇓

classes/modules following SRP are highly cohesive.

Explanation: In the above image, we can see that in low cohesion only one class is responsible to execute lots of jobs that are not in common which reduces the chance of reusability and maintenance. But in high cohesion, there is a separate class for all the jobs to execute a specific job, which results in better usability and maintenance.

Advantages of High Cohesion

→ readability ↑, maintainability ↑, easy to understand/read, etc
  ↳ separation of concerns (single responsibility)

→ easy to debug and make changes
  ↳ have to change less number of modules.

→ easy to test the code

→ reusability ↑, redundancy ↓

Coupling?

What?

⇒ Lower coupling depicts mutual independence of two or more modules or components

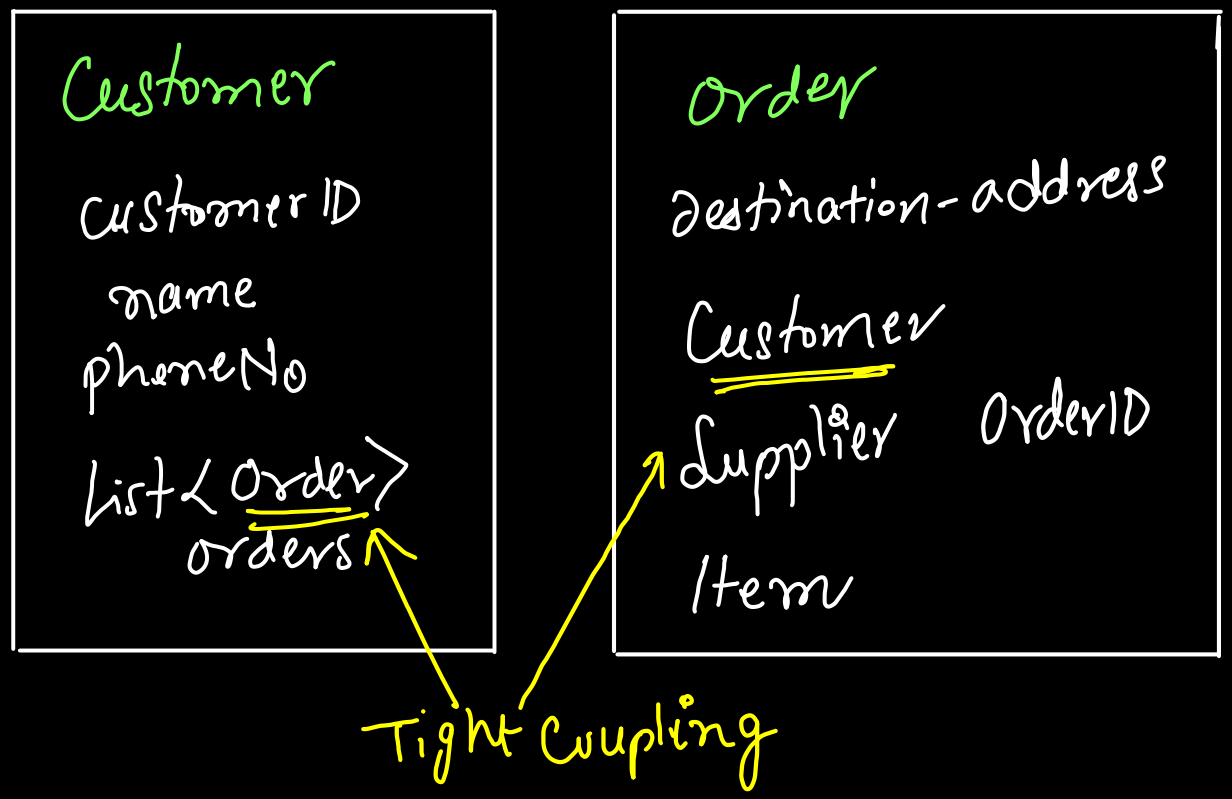
⇒ It is the probability that module B will "break" after an unknown change in "A".

How?

- Eliminate, minimise, and reduce complexity of necessary relationships.
- By hiding implementation details, coupling is reduced. (abstract classes/interface)
- Apply the Law of Demeter.

Tight Coupling ⇒ Two modules are closely related to each other.

Eg Concrete classes storing references of each other & calling each other's methods

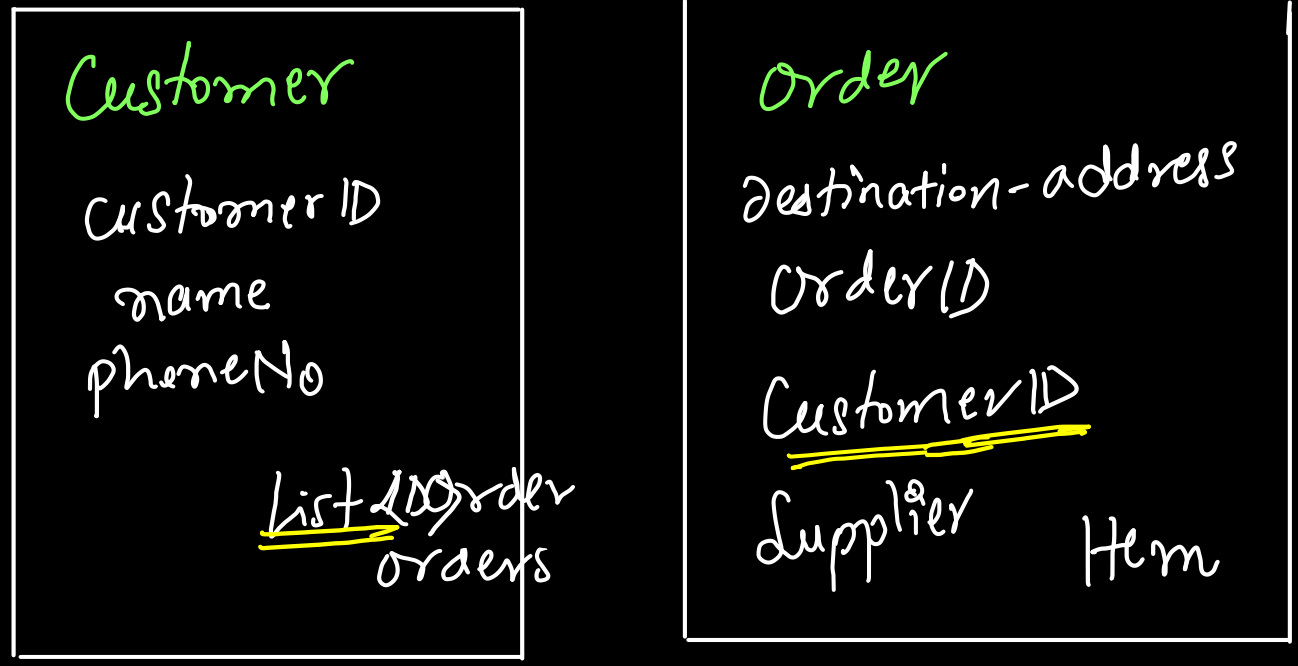


Adding orders will affect Customer class (add order in list of orders)

↳ unnecessary dependency

Loose Coupling

↳ Instead of Customer object itself, we can only have a customerID property in order class.
and similarly vice-versa



⇒ Customer & Order are now independent of each other.

Advantages of Loose Coupling

→ tightly couped modules are difficult to manage, test, etc whereas loosely coupled system is easy to develop & manage.

→ Reusability in tightly coupled system is less whereas it is more (DRY) in loose coupling.

→ Dependent modules' code breaks by change in any of the module, whereas it remains intact for independent modules.

	Cohesion	Coupling
1	Cohesion is the degree to which the elements inside a module belong together.	Coupling is the degree of interdependence between the modules.
2	A module with high cohesion contains elements that are tightly related to each other and united in their purpose.	Two modules have high coupling (or tight coupling) if they are closely connected and dependent on each other.
3	A module is said to have low cohesion if it contains unrelated elements.	Modules with low coupling among them work mostly independently of each other.
4	Highly cohesive modules reflect higher quality of software design	Loose coupling reflects the higher quality of software design

⑧ Law of Demeter :→ "Don't talk to Strangers"

What object should never know the internal details of other objects.

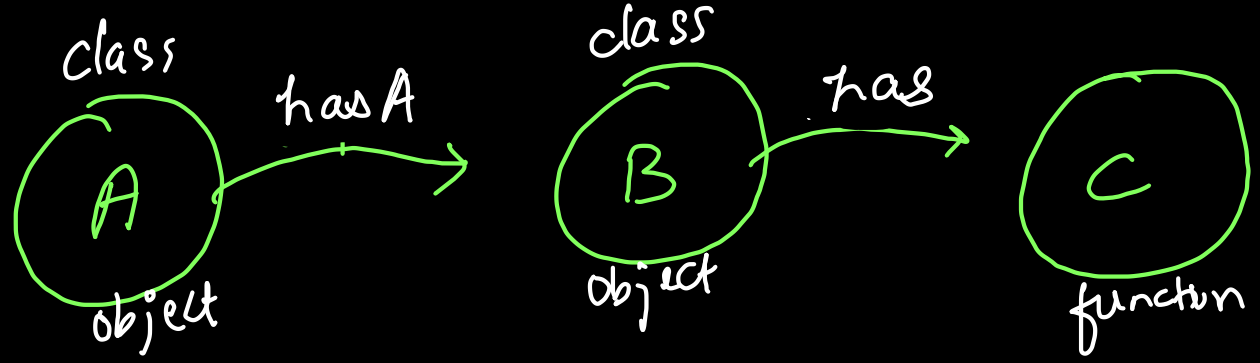
Why It helps to develop loosely coupled modules

How

A method of an object may only call methods of:

1. The object itself.
2. An argument of the method.
3. Any object created within the method.
4. Any direct properties/fields of the object.

Eg



A obj = new A();

obj. Bobj. C fun(); ⇒ violatn of law of Demeter

Car car = new Car();

car. engine. start(); ⇒ not clean code

(9) Composition over Inheritance

Why?

→ Loose coupling b/w classes { non blood relationship } in Composition

→ Tight coupling in inheritance usually breaks code.

→ To avoid wrong inheritance (avoid violating LSP)

How?

→ Apply inheritance (is-A) only if Liskov substitution principle is followed !

→ Compose when there is "has A" relationship or "part of" relationship.

Advantages of Composition over Inheritance

→ To overcome the restriction of multiple inheritance in Java.

eg Reader Writer Application composing both reader & writer objects.

→ Low coupling leads to easy maintanance & flexibility without breaking code.

eg Overriding in Inheritance might break the code if not done properly.

→ Easy unit testing in composition

→ Design Patterns: Strategy, Decorator, Chain of responsibility, Bridge, adapter, etc.

Comparing Inheritance VS Association (or) IS-A relationship with has-A relationship in Java

IS-A relationship
Inheritance

→ using "extends" keyword

→ tightly coupled classes relationship
(change in one class will affect the entire class hierarchy/subclasses)

→ blood relationship

→ Types :→ single level, multilevel, and heirarchial

→ bottom up relationship

has-A relationship
Association

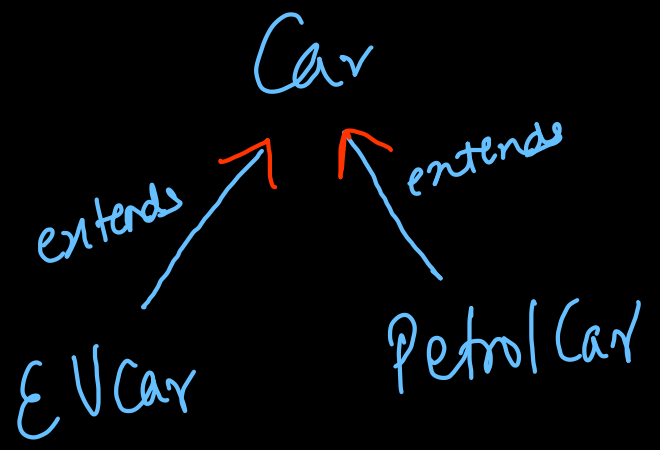
→ using object as data member

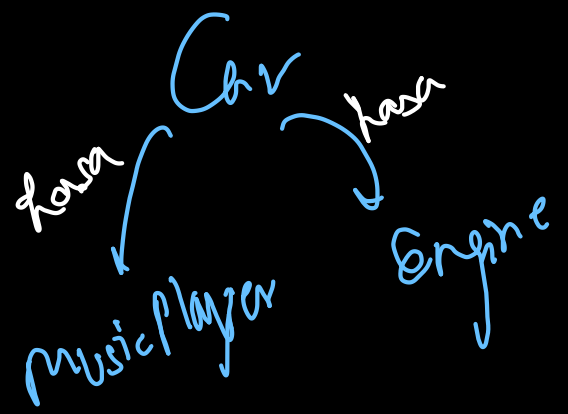
→ loosely couped classes relationship
(change in one class will not directly affect the other class)
↳ access via object only.

→ non blood relationship

→ Types: Aggregation (weak) & Composition (strong)

→ Top Down relationship





& downcasting

EVCar c = new Car();
child parent

implicit upcasting

Car c = new EVCar();
parent child

Inheritance should only be used when:

1. Both classes are in the same logical domain
2. The subclass is a proper subtype of the superclass
3. The superclass's implementation is necessary or appropriate for the subclass
4. The enhancements made by the subclass are primarily additive.

Example 1

```
class Stack extends ArrayList {
    public void push(Object value) { … }
    public Object pop() { … }
}
```

Wrong Inheritance!

"Stack is an Arraylist" is wrong because Stack is not a proper subtype of Arraylist.

→ Stack enforces LIFO policy but this is not enforced by Arraylist class.

→ Although both are linear-sequential data structure, but stack is a queuing concept whereas Arraylist random index

→ Stack's object is bloated. Along with push & pop, stack also gets unnecessary methods like get, set, add, etc.

Stack obj = new Stack();

obj.push()

obj.pop()

~~obj.get(idx)~~

~~obj.addFirst()~~

~~obj.removeLast()~~

LIFO violated

```java
class MyStack{
    List<Integer> stk;

    public MyStack(ArrayList<Integer> arr){
        stk = arr;
    }

    public MyStack(LinkedList<Integer> arr){
        stk = arr;
    }

    public void push(int a){
        stk.add(a);
    }

    public int remove(){
        return stk.remove(stk.size() - 1);
    }
}
```

Examples

WorkingProfessional

↓ extends

SoftwareDeveloper

Student

↓ extends

CollegeStudent

SDEIntern

extends

SoftwareDeveloper

extends

CollegeStudent

SDEIntern

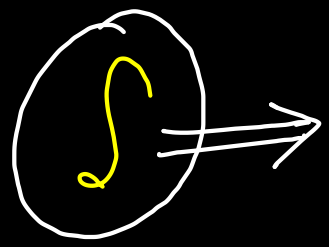
extends / Composes

SoftwareDeveloper

composes

Student

10



S

O

L

I

D

Single Responsibility Principle {S in SOLID}

=

Command - Query Separation

=

Separation of Concerns Principle

=

Curly's law

"A class should have only one reason to change."

or

"A class should have a single responsibility"

⇒ If we have 2 reasons to change for a class, we have to split the functionality in two classes.

It will help to make changes in the corresponding divided class without affecting the other one.

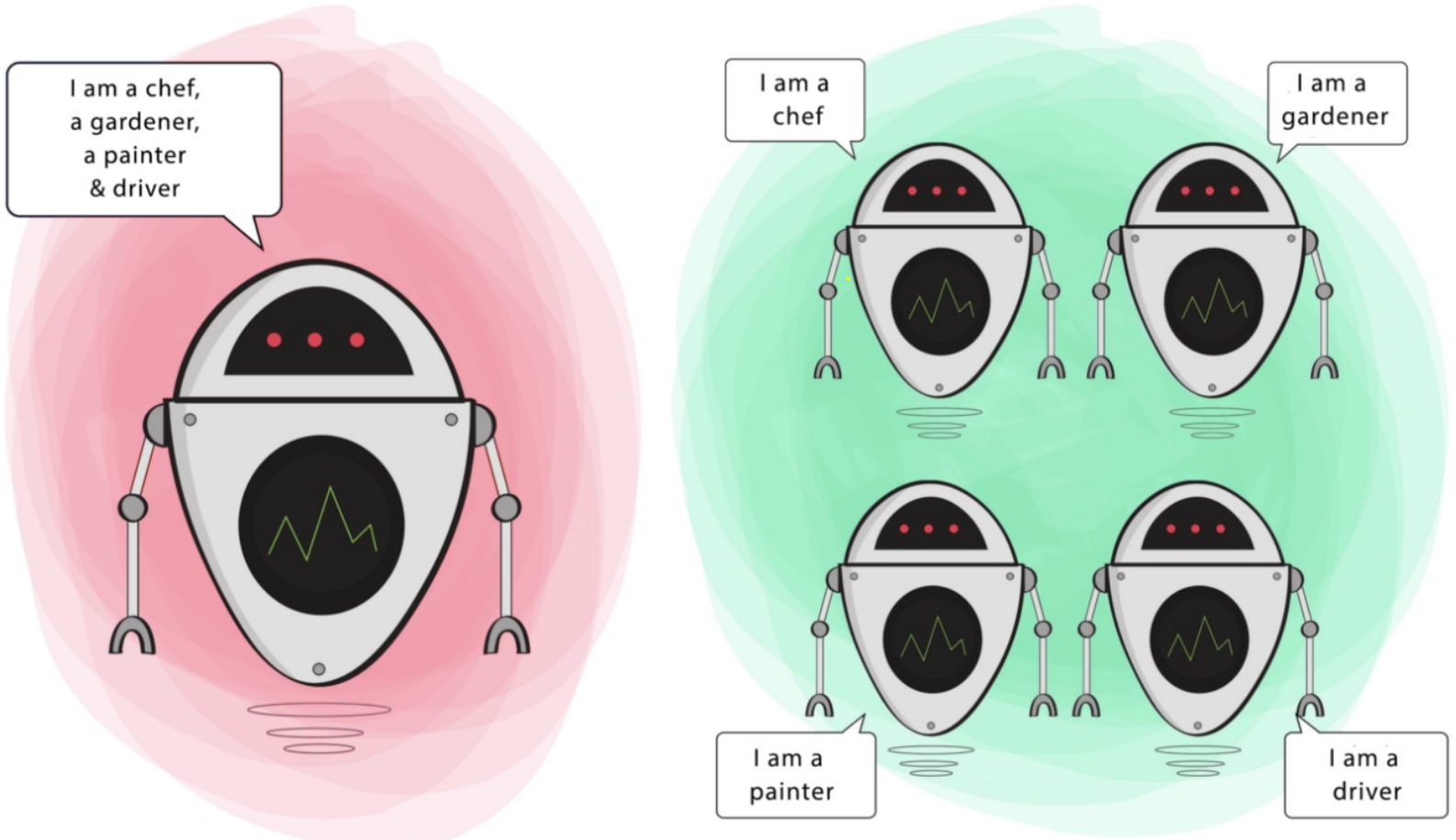
Thus debugging becomes easier as breaking of class B due to change in class A, it's chances will reduce.

Module { set of functions / class / package / service }

Only one Actor  should change it !

↳ external entities representing some roles

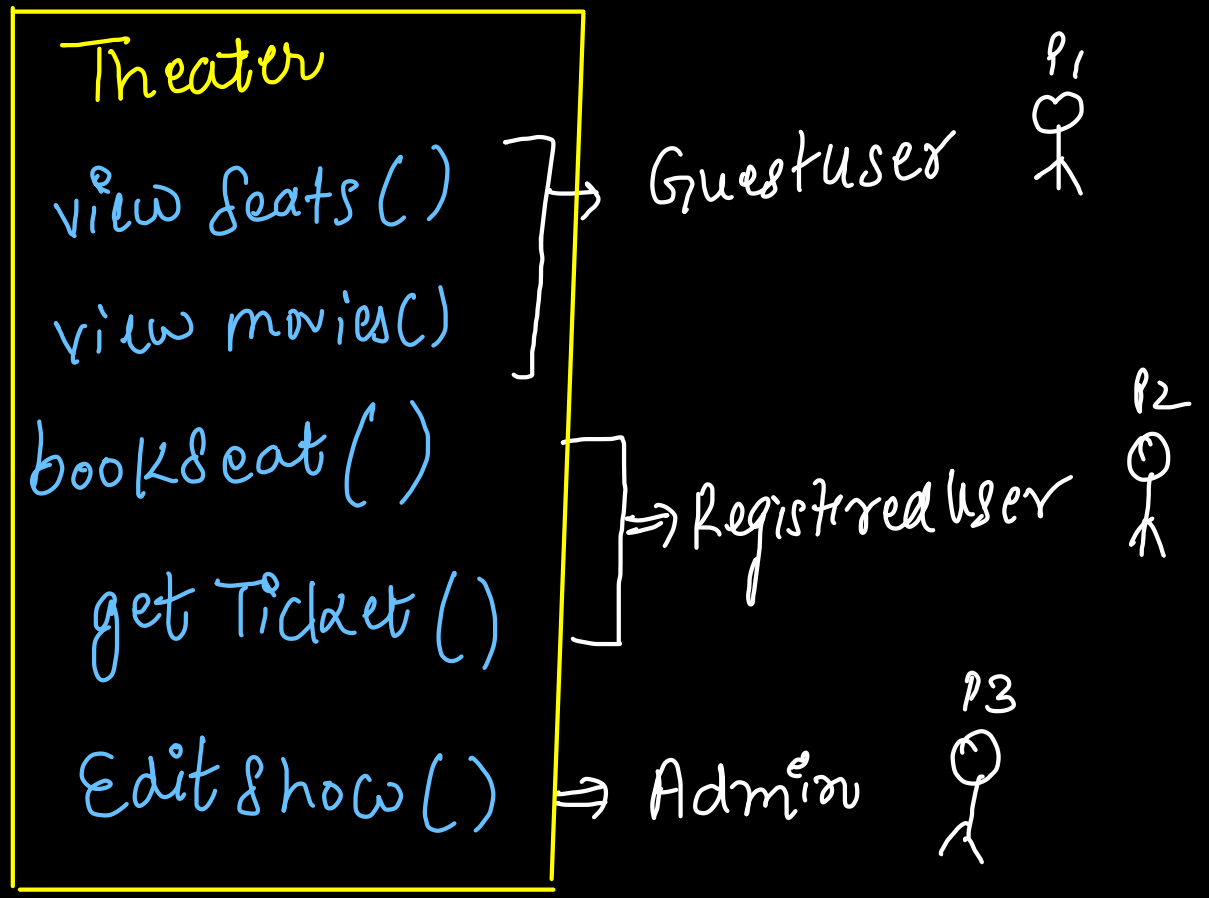
eg human users, admin/staff,
third party API, hardware, etc



Single Responsibility

Example 1

Problem



Violating Single Responsibility Principle!

⇒ change in bookSeat may break viewSeats.

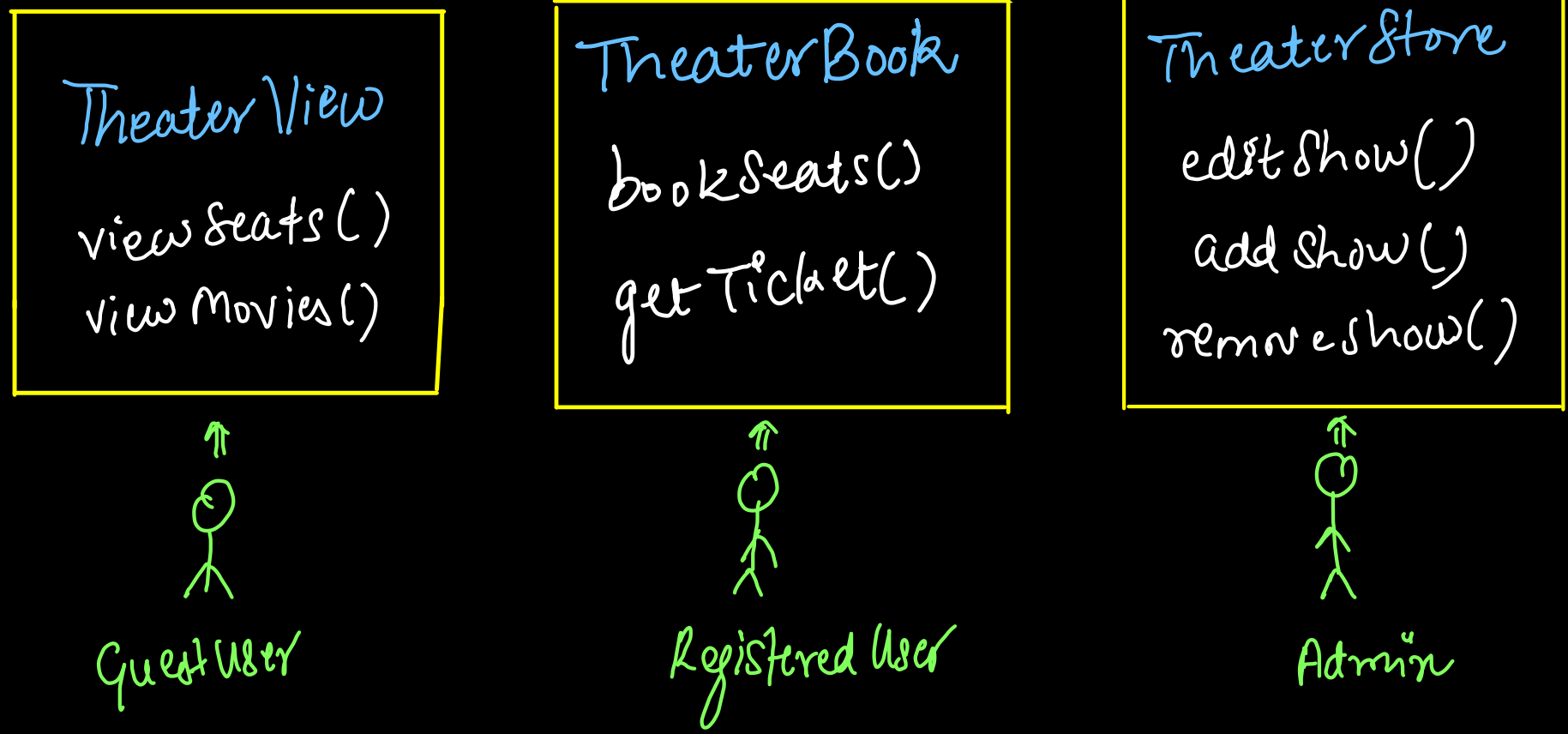
or change in editShow might break bookSeat

```java
class App {
    Run | Debug
    public static void main(String[] args) {
        Theater pvr = new Theater();

        pvr.editShow(new Admin());
        pvr.viewShow(new User());
        pvr.bookShow(new RegUser());
    }
}
```

```java
class Admin {
}

class User {
}

class RegUser {
}

class Theater {
    public void editShow(Admin user) {
        System.out.println(x: "Show Added/Modified in Database");
    }

    public void viewShow(User user) {
        System.out.println(x: "View Show Listings");
    }

    public void bookShow(RegUser user) {
        System.out.println(x: "Select seats and proceed to payments");
    }
}
```

Solution

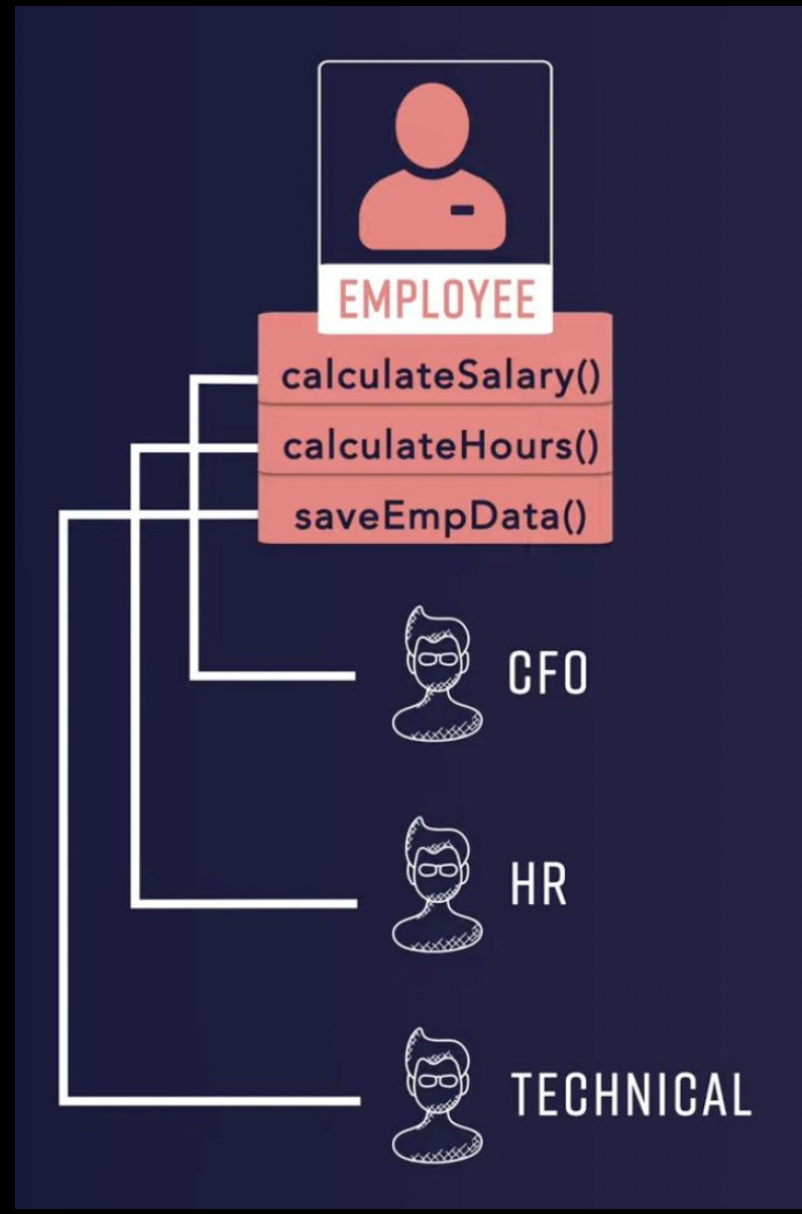
Separation of concerns!



```java
class TheaterStore {
    public void editShow(Admin user) {
        System.out.println(x: "Show Added/Modified in Database");
    }
}

class TheaterView {
    public void viewShow(User user) {
        System.out.println(x: "View Show Listings");
    }
}

class TheaterBook {
    public void bookShow(RegUser user) {
        System.out.println(x: "Select seats and proceed to payments");
    }
}

class App {
    Run | Debug
    public static void main(String[] args) {
        TheaterStore pvr1 = new TheaterStore();
        pvr1.editShow(new Admin());

        TheaterView pvr2 = new TheaterView();
        pvr2.viewShow(new User());

        TheaterBook pvr3 = new TheaterBook();
        pvr3.bookShow(new RegUser());
    }
}
```

Example 2

Problem



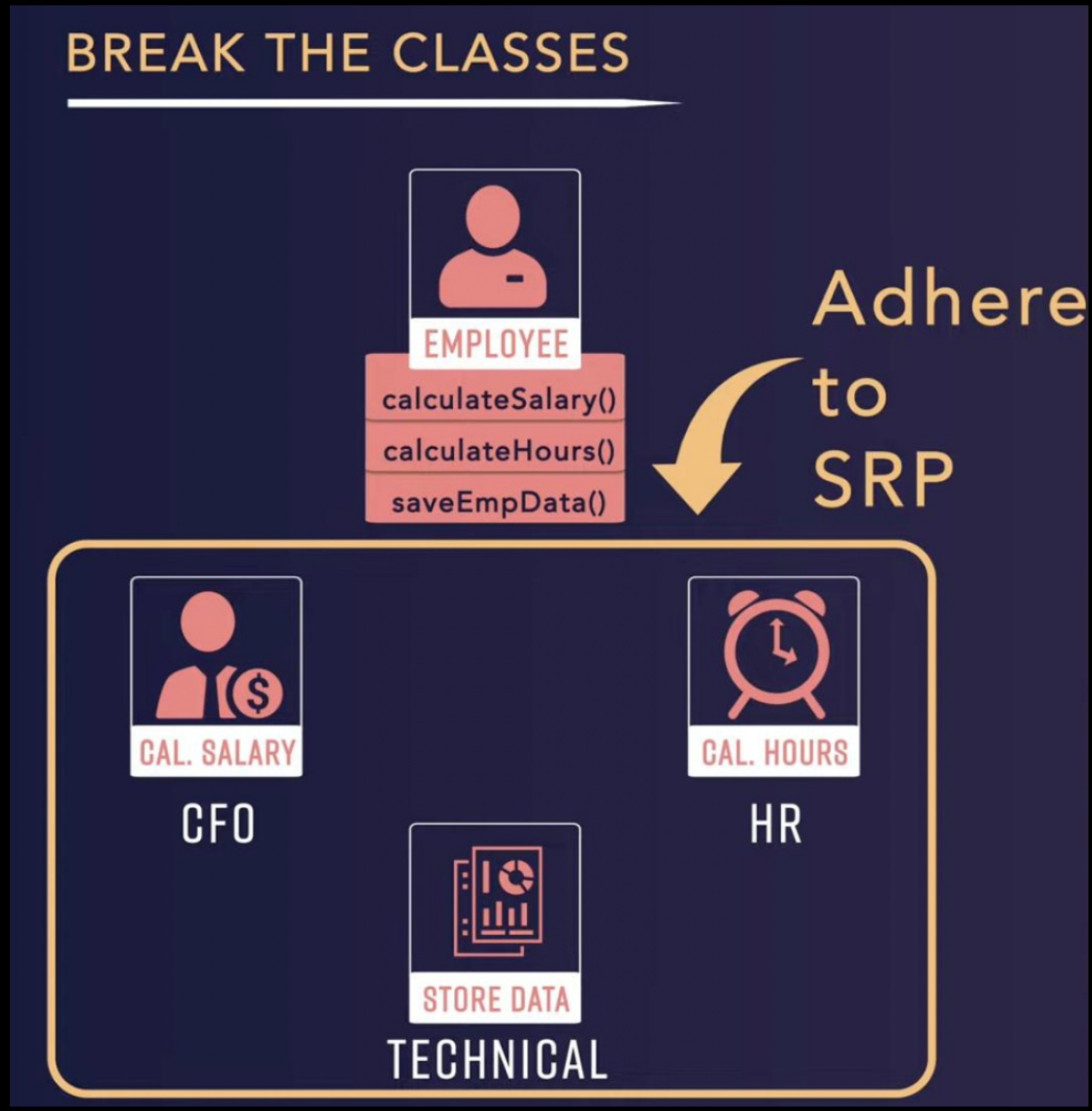
clean code book example

Single Employee class is handling responsibilities of CFO, HR, Technical

↓

violates SRP

Solution


BREAK THE CLASSES
EMPLOYEE
calculateSalary()
calculateHours()
saveEmpData()
Adhere to SRP
CAL. SALARY
CFO
CAL. HOURS
HR
STORE DATA
TECHNICAL

11 Command - Query Separation

Query → returns state without modifying it. { Getters } → return type

Command → changes/modifies state without returning it. { Setter } → void

Violation :→ Queue (Java) → remove → delete (state modify)
return (return change)

What?

The Command Query Separation principle states that each method should be either a command that performs an action or a query that returns data to the caller but not both. Asking a question should not modify the answer.

Why?

With this principle applied the programmer can code with much more confidence. The query methods can be used anywhere and in any order since they do not mutate the state. With commands one has to be more careful.

how?

- Implement each method as either a query or a command
- Apply naming convention to method names that implies whether the method is a query or a command

Curly's law

Curly's Law is about choosing a single, clearly defined goal for any particular bit of code: Do One Thing.

Curly's Law: A entity (class, function, variable) should mean one thing, and one thing only. It should not mean one thing in one circumstance and carry a different value from a different domain some other time. It should not mean two things at once. It should mean One Thing and should mean it all of the time.

Separation of Concerns

Separation of concerns is a design principle for separating a computer program into distinct sections, such that each section addresses a separate concern. For example the business logic of the application is a concern and the user interface is another concern. Changing the user interface should not require changes to business logic and vice versa.

(12)

S
O
L
I
D

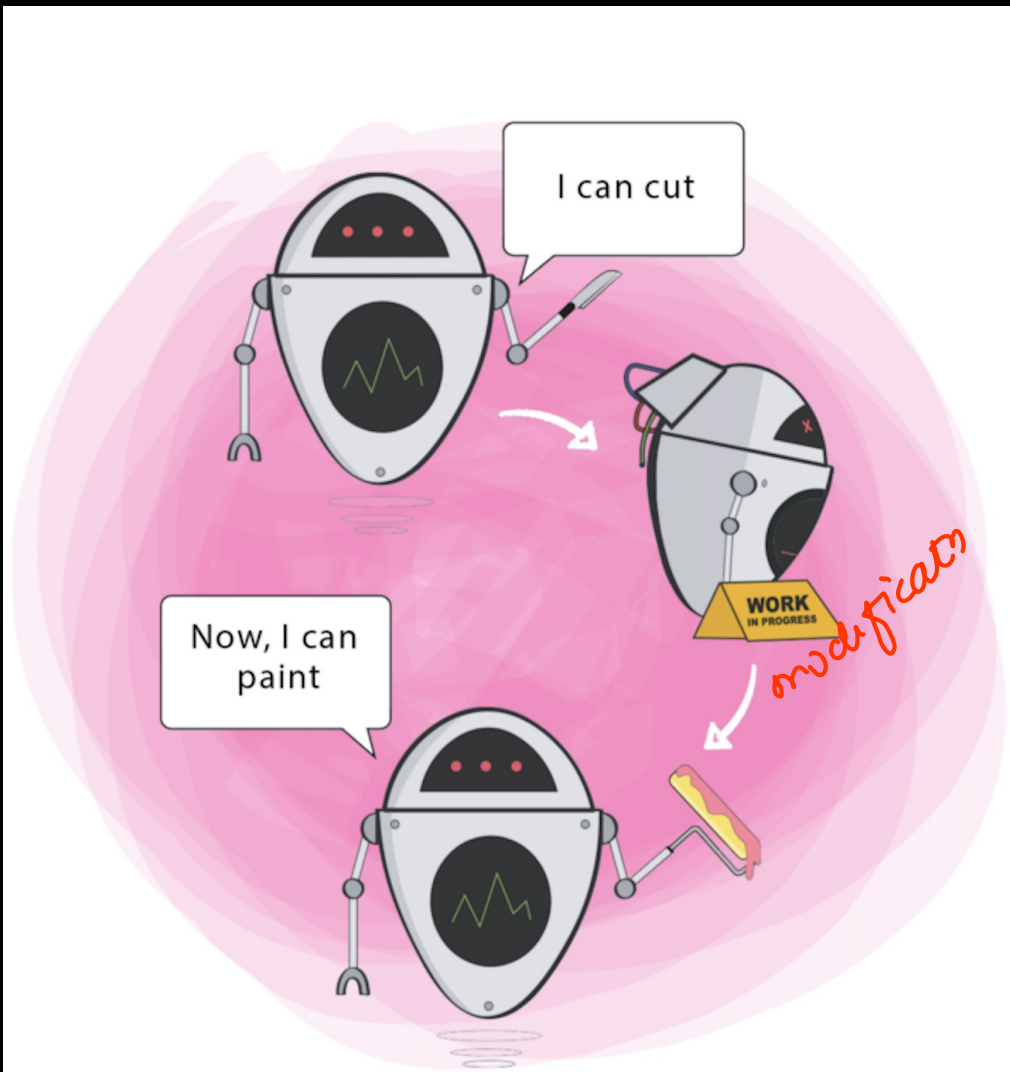
Open - Closed Principle

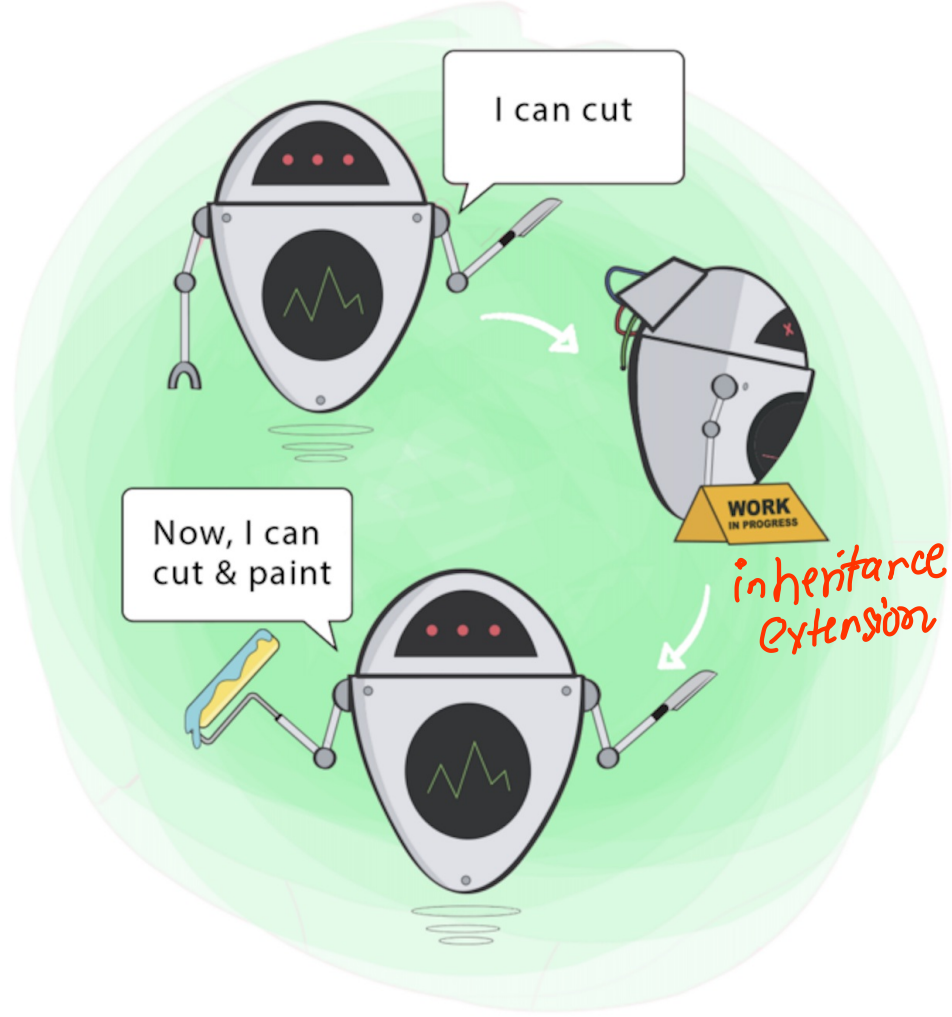
What → modules (eg class, service, package) should be open for extension, but closed for modification.

Why → classes modification might break the class hierarchy tree, become less readable & maintainable.
↳ minimum classes get affected by a change in code.

How → use inheritance, polymorphism & abstraction

- inheritance → extending classes
- polymorphism → polymorphic variable
- abstraction → hiding unnecessary details


I can cut
WORK
IN PROGRESS
Now, I can
paint
modification


I can cut
WORK
IN PROGRESS
Now, I can
cut & paint
inheritance
extension

Open-Closed

Example 1

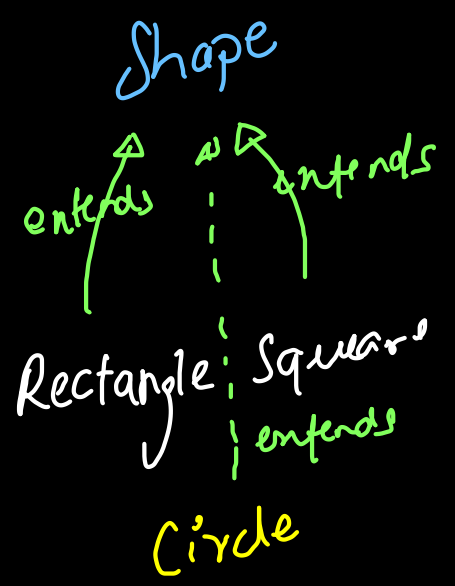
Problem

```
class Shape {
    private int shapeID;

    public int getShapeID() {
        return shapeID;
    }

    public void setShapeID(int shapeID) {
        this.shapeID = shapeID;
    }

}

class Rectangle extends Shape {
    int length, breadth;

    public Rectangle(int length, int breadth) {
        super.setShapeID(shapeID: 1);
        this.length = length;
        this.breadth = breadth;
    }
}

class Square extends Shape {
    int side;

    public Square(int side) {
        super.setShapeID(shapeID: 2);
        this.side = side;
    }
}
```

Violates Open Closed Principle

```
class GraphEditor {
    public static void drawShape(Shape shape) {
        if (shape.getShapeID() == 1) {
            System.out.println(x: "Draw a rectangle");

            int length = ((Rectangle) shape).length;
            int breadth = ((Rectangle) shape).breadth;

            System.out.println("of length = " + length);
            System.out.println(" and breadth = " + breadth);

        } else if (shape.getShapeID() == 2) {
            System.out.println(x: "Draw a Square");

            int side = ((Square) shape).side;
            System.out.println("of side = " + side);
        }
    }

    Run | Debug
    public static void main(String[] args) {
        List<Shape> shapes = new ArrayList<>();

        shapes.add(new Rectangle(length: 10, breadth: 20));
        shapes.add(new Square(side: 5));

        for (Shape shape : shapes) {
            drawShape(shape);
        }
    }
}
```

else if (shape.getID() == 3) { // draw circle }



```java
abstract class Shape {
    public abstract void drawShape();
}

class Rectangle extends Shape {
    int length, breadth;

    public Rectangle(int length, int breadth) {
        this.length = length;
        this.breadth = breadth;
    }

    @Override
    public void drawShape() {
        System.out.println(x: "Draw a rectangle");
        System.out.println("of length = " + length);
        System.out.println(" and breadth = " + breadth);
    }
}

class Square extends Shape {
    int side;

    public Square(int side) {
        this.side = side;
    }

    @Override
    public void drawShape() {
        System.out.println(x: "Draw a Square");
        System.out.println("of side = " + side);
    }
}
```

Solution

```java
class GraphEditor {
    Run | Debug
    public static void main(String[] args) {
        List<Shape> shapes = new ArrayList<>();

        shapes.add(new Rectangle(length: 10, breadth: 20));
        shapes.add(new Square(side: 5));

        for (Shape shape : shapes) {
            shape.drawShape();
        }
    }
}
```

this is "open for extension"
and "closed for modification"

Example 2

↓

Problem

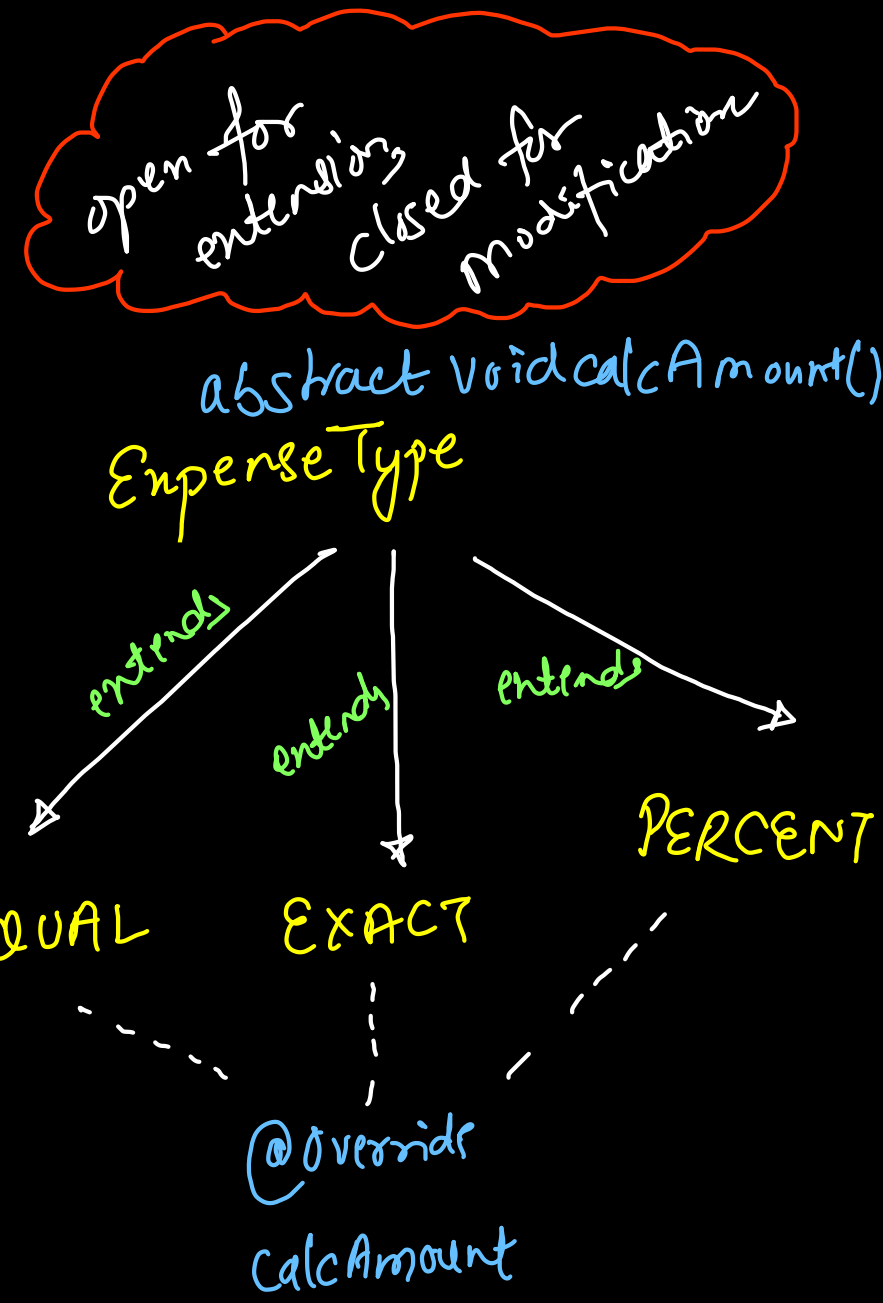
Splitwise App : ExpenseType = EQUAL, PERCENT, EXACT

```
calcExpense(){
    if(expensetype == EQUAL)
        // equally split among debtors
    else if (expensetype == PERCENT)
        // split in fractions among debtors
    else // split in exact amounts specified!
}
```

→ open for modification

Solution

```java
public enum ExpenseType {
    EQUAL {
        @Override
        public double calcAmount(double amount, double countSplit) {
            return amount / countSplit;
        }
    },
    EXACT {
        @Override
        public double calcAmount(double amount, double exactSplit) {
            return exactSplit;
        }
    },
    PERCENT {
        @Override
        public double calcAmount(double amount, double percentSplit) {
            return (amount * percentSplit) / 100.0;
        }
    };

    public abstract double calcAmount(double totalAmount, double split);

    public boolean validate(double totalAmount, List<Split> splits) {
        double sumSplit = 0.0;
        for (Split split : splits) {
            sumSplit += split.getAmount();
        }

        return ((sumSplit - totalAmount) <= 0.1);
    }
}
```



Example 3) Problem

```
public class InvoicePersistence {
    Invoice invoice;

    public InvoicePersistence(Invoice invoice) {
        this.invoice = invoice;
    }

    public void saveToFile(String filename) {
        // Creates a file with given name and writes the invoice
    }

    public void saveToDatabase() {
        // Saves the invoice to database
    }
}
```

Unfortunately we, as the lazy developer for the book store, did not design the classes to be easily extendable in the future. So in order to add this feature, we have modified the **InvoicePersistence** class.

If our class design obeyed the Open-Closed principle we would not need to change this class.

Example 3) Solution

```
interface InvoicePersistence {

    public void save(Invoice invoice);
}
```

```
public class DatabasePersistence implements InvoicePersistence {

    @Override
    public void save(Invoice invoice) {
        // Save to DB
    }
}
```

```
public class FilePersistence implements InvoicePersistence {

    @Override
    public void save(Invoice invoice) {
        // Save to file
    }
}
```

S O L I D

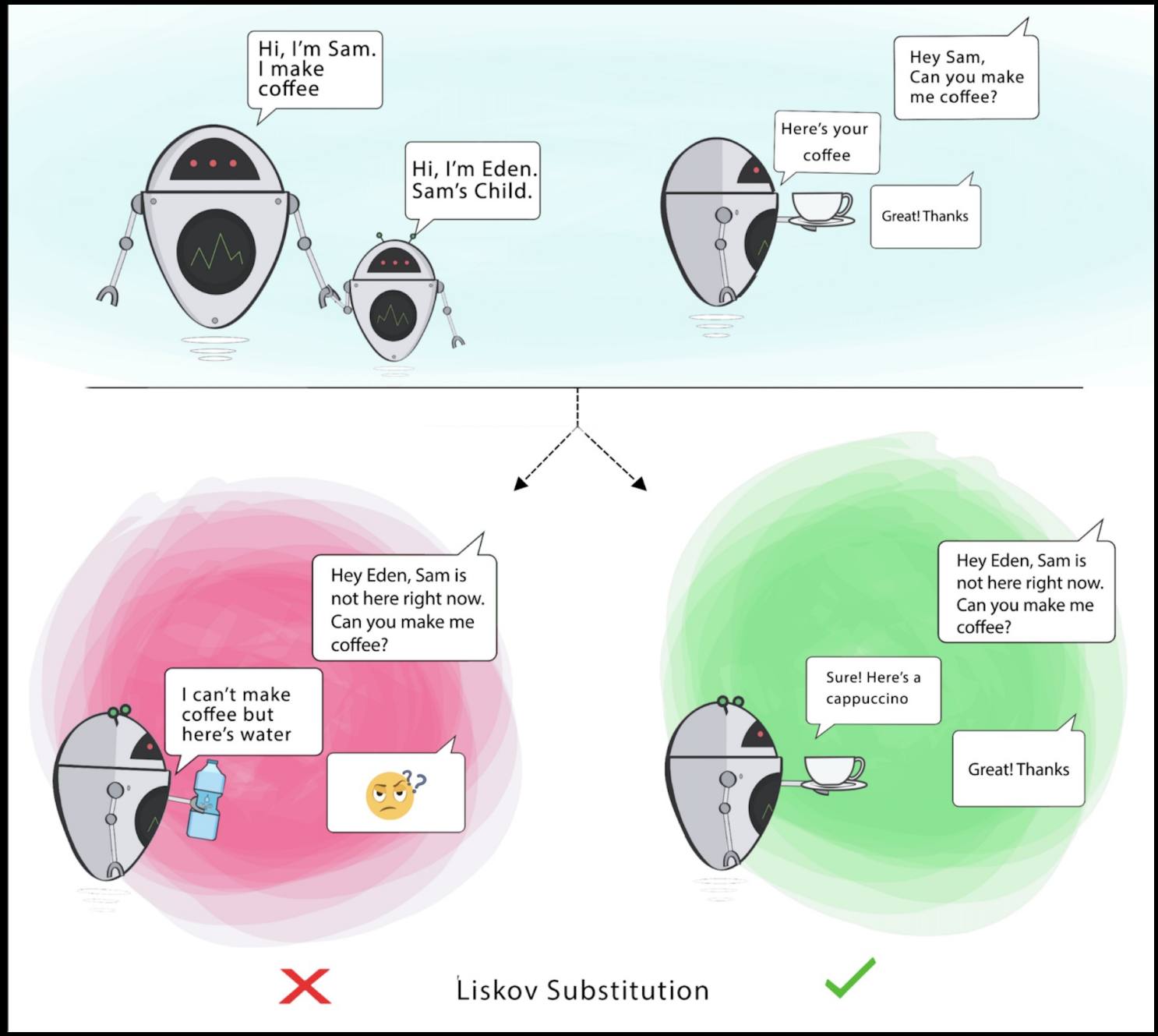
(13) Liskov - Substitution Principle

"Derived types must be completely substitutable for their base types."

⇒ We should make sure that new derived classes are extending the base classes without changing their behavior.

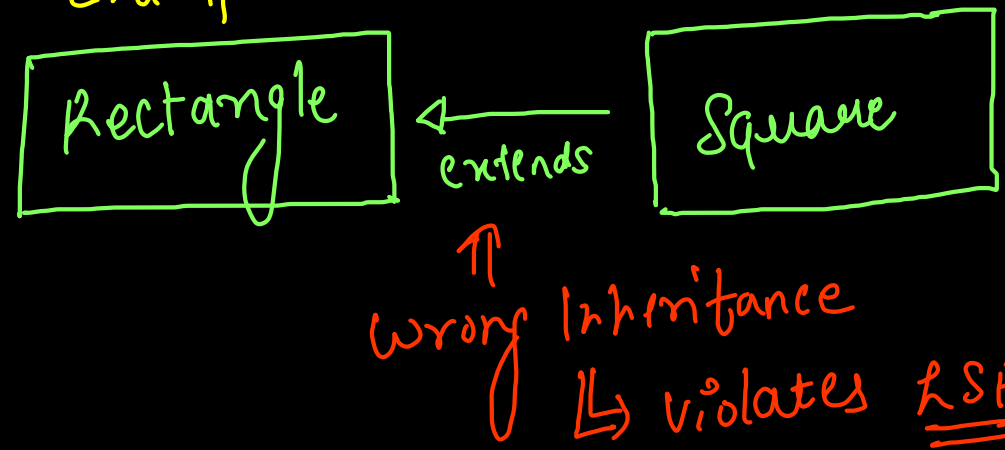
"Let f(x) be a property provable about objects of Type T. Then f(y) should be true for objects y of Type S where S is a subtype of T."

⇒ If functions takes an instance of a class, the same function should also be take instance of any of the derived classes.


Hi, I'm Sam. I make coffee
Hi, I'm Eden. Sam's Child.
Hey Sam, Can you make me coffee?
Here's your coffee
Great! Thanks
Hey Eden, Sam is not here right now. Can you make me coffee?
I can't make coffee but here's water
Hey Eden, Sam is not here right now. Can you make me coffee?
Sure! Here's a cappuccino
Great! Thanks
Liskov Substitution

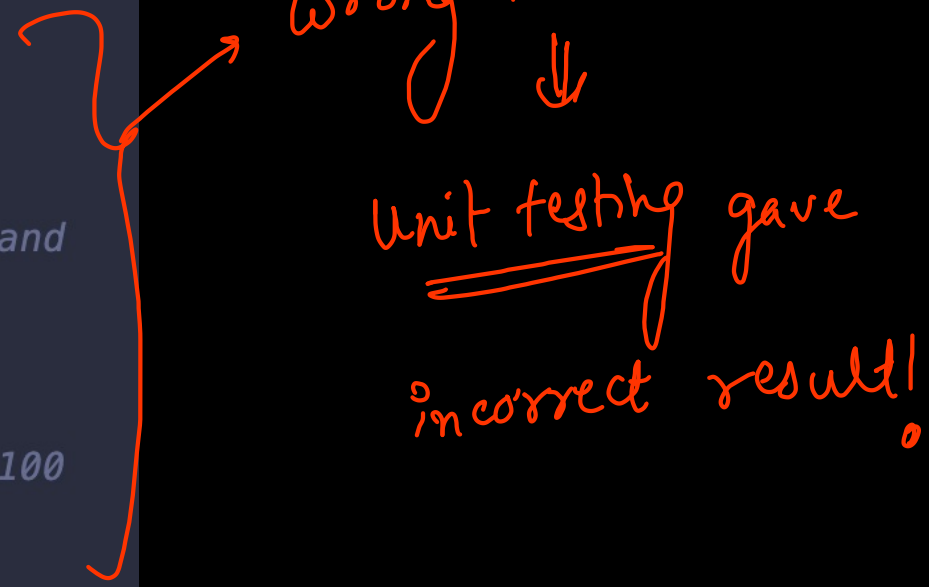
```java
class Rectangle {
    protected int width;
    protected int height;

    public void setWidth(int width) {
        this.width = width;
    }

    public void setHeight(int height) {
        this.height = height;
    }

    public int getWidth() {
        return width;
    }

    public int getHeight() {
        return height;
    }

    public int getArea() {
        return width * height;
    }
}
```

Example 1 : Problem




```java
class Square extends Rectangle {
    public void setWidth(int width) {
        super.width = width;
        super.height = width;
    }

    public void setHeight(int height) {
        super.width = height;
        super.height = height;
    }
}
```

```
class Factory {
    private static Rectangle getNewRectangle() {
        // it can be an object returned by some factory ...
        return new Square();
    }

    Run | Debug
    public static void main(String args[]) {
        Rectangle r = Factory.getNewRectangle();

        r.setWidth(width: 5);
        r.setHeight(height: 10);
        // user knows that r it's a rectangle.
        // It assumes that he's able to set the width and
        height as for the base class

        System.out.println(r.getArea());
        // now he's surprised to see that the area is 100
        instead of 50.
    }
}
```



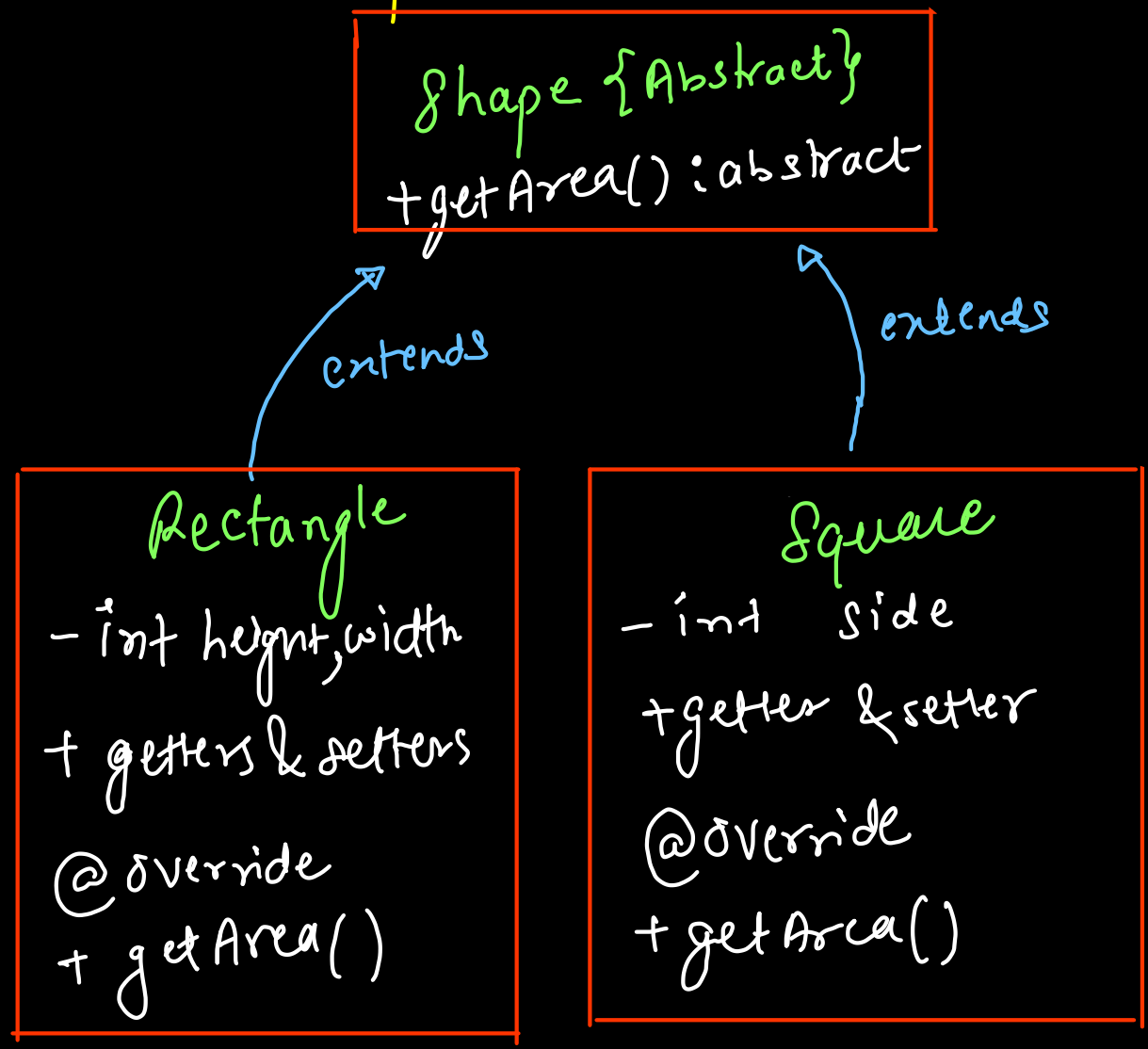
Example 1: Solution 1

Inheritance & Composition ✓

```
class Square {
    private Rectangle rect = new Rectangle();

    public void setSide(int side) {
        rect.setHeight(side);
        rect.setWidth(side);
    }

    public int getWidth() {
        return rect.getWidth();
    }

    public int getHeight() {
        return rect.getHeight();
    }

    public int getArea() {
        return this.getWidth() * this.getHeight();
    }
}
```

```
class Factory {
    Run | Debug
    public static void main(String args[]) {
        Square sq = new Square();
        sq.setSide(side: 5);

        System.out.println(sq.getArea());
    }
}
```

Example 1 : Solution2




```java
abstract class Shape {
    public abstract int getArea();
}

class Rectangle extends Shape {
    private int width, height;

    public void setWidth(int width) {
        this.width = width;
    }

    public void setHeight(int height) {
        this.height = height;
    }

    public int getWidth() {
        return width;
    }

    public int getHeight() {
        return height;
    }

    @Override
    public int getArea() {
        return getHeight() * getWidth();
    }
}
```

```java
class Square extends Shape {
    private int side;

    public int getSide() {
        return side;
    }

    public void setSide(int side) {
        this.side = side;
    }

    @Override
    public int getArea() {
        return getSide() * getSide();
    }
}
```

```java
class Factory {
    private static Shape getShape() {
        // it can be an object returned by some factory ...
        return new Square();
    }

    Run | Debug
    public static void main(String args[]) {
        Shape r = Factory.getShape();

        if (r instanceof Rectangle) {
            ((Rectangle) r).setWidth(width: 5);
            ((Rectangle) r).setHeight(height: 10);
        } else if (r instanceof Square) {
            ((Square) r).setSide(side: 15);
        }

        System.out.println(r.getArea());
    }
}
```

Example 2 :-

```java
class Product {
    private double price, gstPercent;

    Product(double price, double gstPercent) {
        this.price = price;
        this.gstPercent = gstPercent;
    }

    public double getPriceWithTax() {
        return price + price * (gstPercent / 100.0);
    }

    public double getPriceWithoutTax() {
        return price;
    }

    public double getGstPercent() throws Exception {
        return gstPercent;
    }

    public void setGstPercent(double gstPercent) throws
    Exception {
        this.gstPercent = gstPercent;
    }
}
```

Product ← extends — Dairy

Parent — child

↓

wrong Inheritance

↳ violates LSP

gstfree

```java
class Dairy extends Product {
    Dairy(double price) {
        super(price, gstPercent: 0.0);
    }

    @Override
    public double getGstPercent() throws Exception {
        throw new Exception(message: "No GST on Dairy");
    }

    @Override
    public void setGstPercent(double gstPercent) throws
    Exception {
        throw new Exception(message: "No GST on Dairy");
    }
}
```

```java
class Driver {
    public static void helper(Product p) {
        try {
            System.out.println(p.getGstPercent());
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }

        System.out.println(p.getPriceWithTax());
        System.out.println(p.getPriceWithoutTax());
    }

    Run | Debug
    public static void main(String[] args) {
        Product phone = new Product(price: 20000.0,
        gstPercent: 18);
        helper(phone);

        Product milk = new Dairy(price: 100.0);
        helper(milk);
    }
}
```

Output:→

```
18.0
23600.0
20000.0
No GST on Dairy
100.0
100.0
```

Solution

```java
class ProductWithoutGst {
    protected double price;

    ProductWithoutGst(double price) {
        this.price = price;
    }

    public double getPriceWithoutTax() {
        return price;
    }
}
```

```java
class Dairy extends ProductWithoutGst {
    Dairy(double price) {
        super(price);
    }
}
```

```java
class Product extends ProductWithoutGst {
    protected double gstPercent;

    Product(double price, double gstPercent) {
        super(price);
        this.gstPercent = gstPercent;
    }

    public double getPriceWithTax() {
        return price + price * (gstPercent / 100.0);
    }

    public double getGstPercent() {
        return gstPercent;
    }

    public void setGstPercent(double gstPercent) {
        this.gstPercent = gstPercent;
    }
}
```

```java
class Driver {
    public static void helper(ProductWithoutGst p) {
        if (p instanceof Product) {
            System.out.println(((Product) p).getGstPercent());
            System.out.println(((Product) p).getPriceWithTax());
        } else {
            System.out.println(p.getPriceWithoutTax());
        }
    }

    Run | Debug
    public static void main(String[] args) {
        ProductWithoutGst phone = new Product(price: 20000.0,
        gstPercent: 18);
        helper(phone);

        ProductWithoutGst milk = new Dairy(price: 100.0);
        helper(milk);
    }
}
```

output

```
18.0
23600.0
100.0
```

LSP Summary :→

Parent

extends

Child

overridden functionality

meaningful over-ridded functionality

no such function in parent

a new functionality in child

functionality not overrided

functionality of parent supported in child

functionality not overrided

violatn of LSP

functionality of parent not supported in child

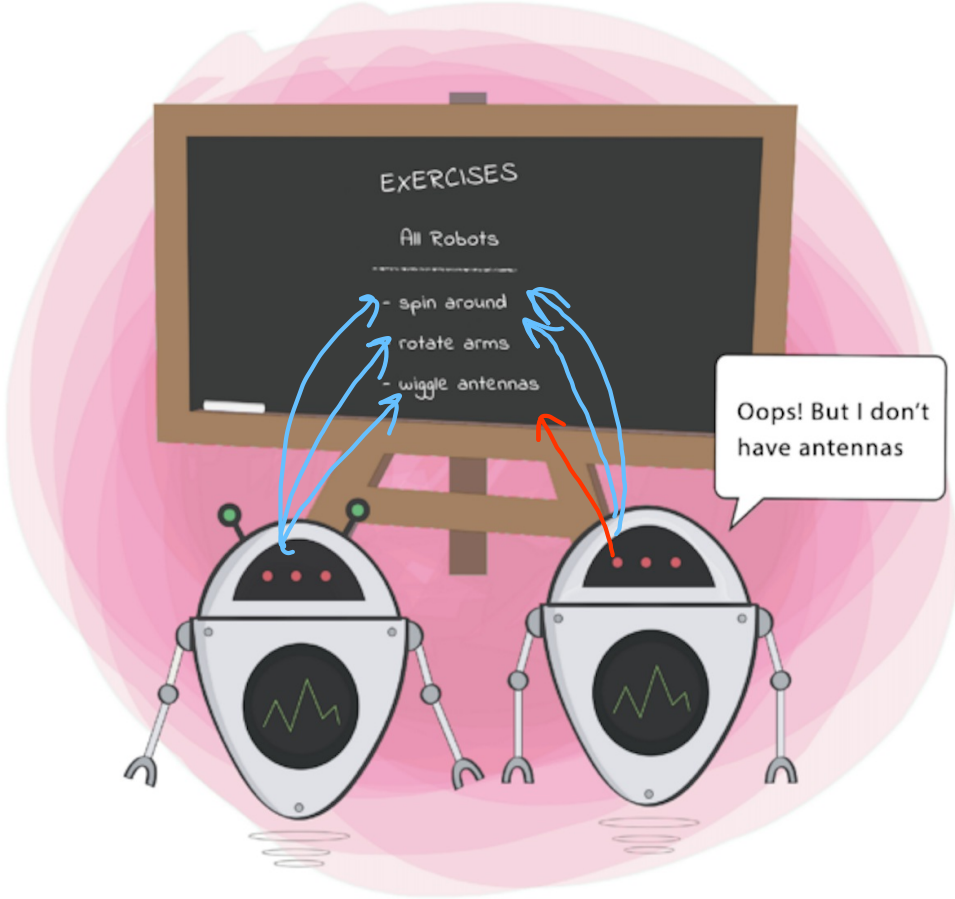
S O L I D

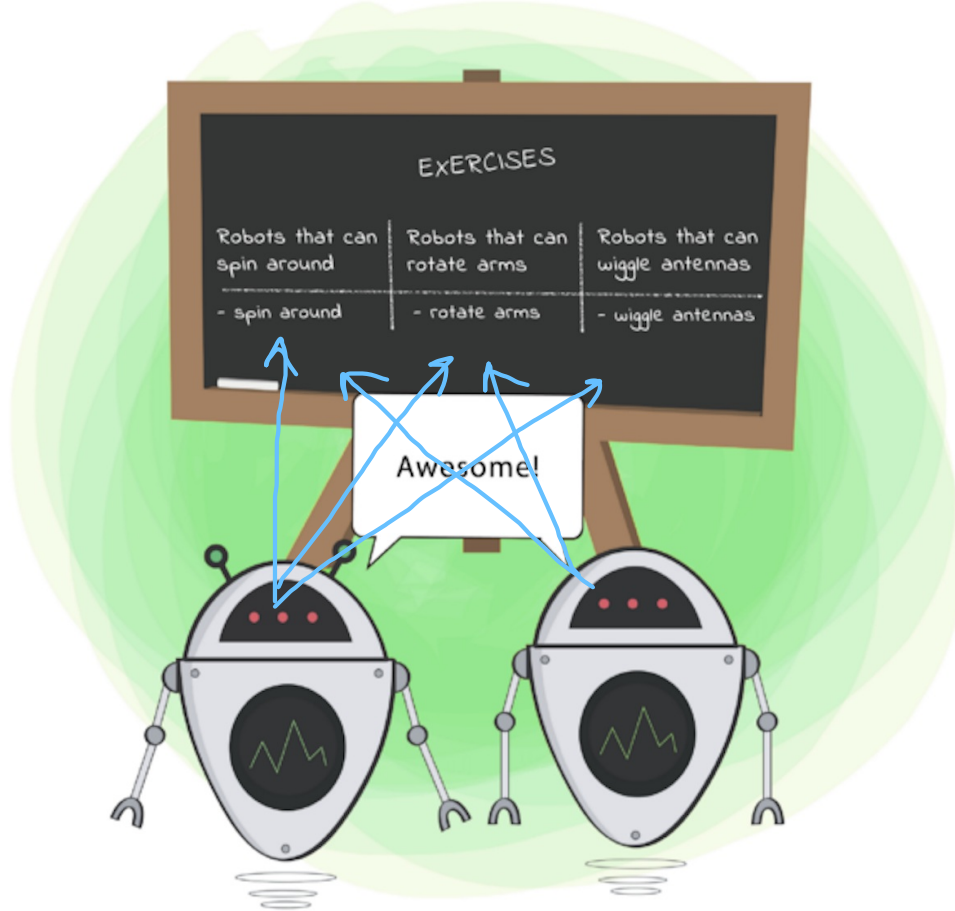
(14) Interface - Segregation Principle

"Clients should not be forced to depend upon interfaces that they don't use"

⇒ avoid creating fat/bloat interfaces. Single interface should not have contract of different responsibilities.

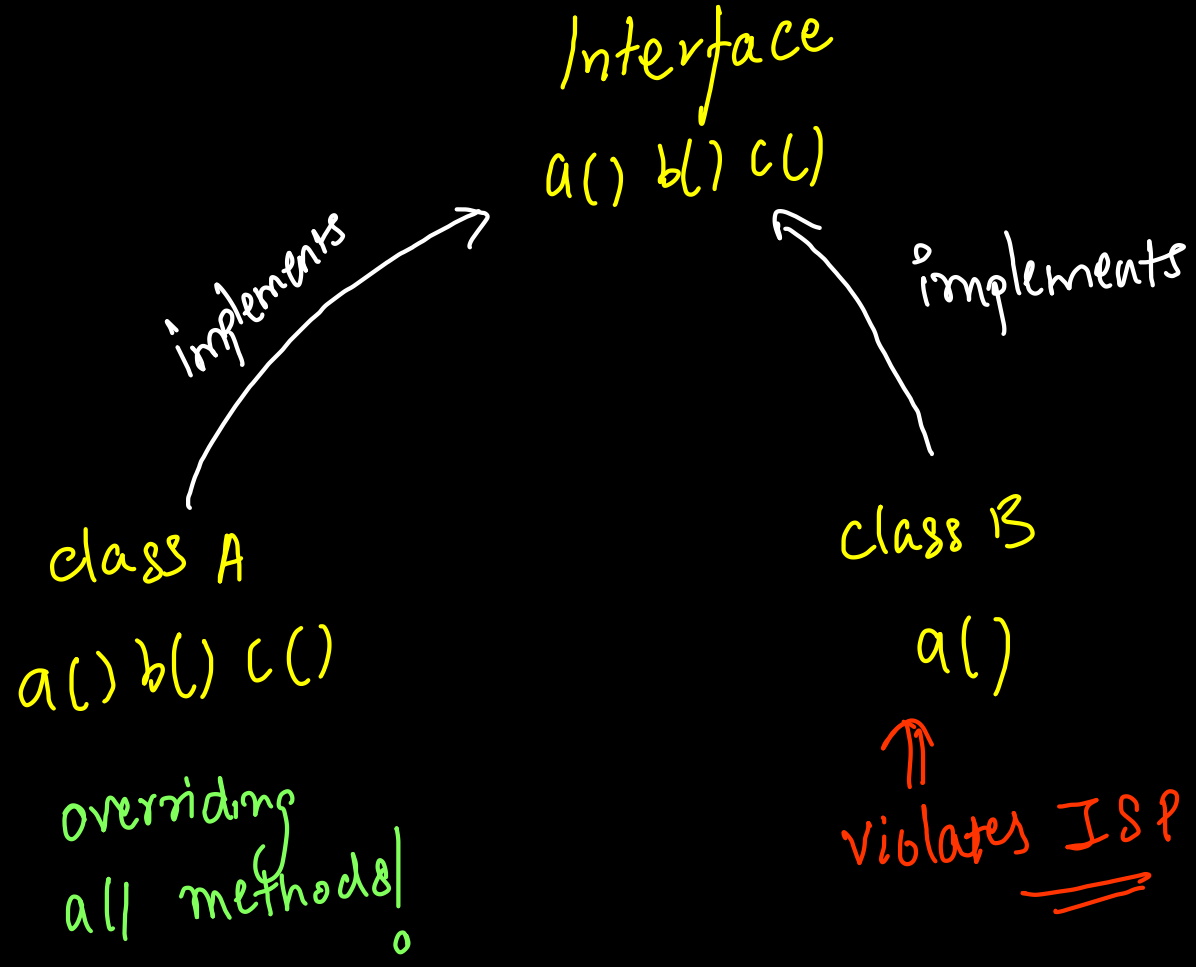
Instead, multiple interfaces should be there having contracts of single responsibility.





Interface Segregation

⇒ Design interfaces so that classes implementing them do not have to override useless methods.



Example 1) Problem

```java
interface Worker {
    public void work();

    public void eat();
}

class Human implements Worker {
    public void work() {
        System.out.println(x: "Slow Work");
    }

    public void eat() {
        System.out.println(x: "Eat Food");
    }
}

class Robot implements Worker {
    public void work() {
        System.out.println(x: "Fast Work");
    }

    public void eat() {
        // Do Nothing Function
    }
}
```

```java
class Manager {
    Run | Debug
    public static void main(String[] args) {
        Worker human = new Human();
        human.work();
        human.eat();

        Worker robot = new Robot();
        robot.work();
        robot.eat();
    }
}
```

Example 1) Solution

```java
interface Worker {
    public void work();
}

interface Eater {
    public void eat();
}

class Human implements Worker, Eater {
    @Override
    public void work() {
        System.out.println(x: "Slow Work");
    }

    @Override
    public void eat() {
        System.out.println(x: "Eat Food");
    }
}

class Robot implements Worker {
    @Override
    public void work() {
        System.out.println(x: "Fast Work");
    }
}
```

```java
class Manager {
    Run | Debug
    public static void main(String[] args) {
        Human human = new Human();
        human.work();
        human.eat();

        Robot robot = new Robot();
        robot.work();
    }
}
```

Example 2) Problem

```java
interface Shape {
    public double area();

    public double volume();
}

class Cuboid implements Shape {
    int length = 10, breadth = 20, height = 30;

    @Override
    public double area() {
        return 2 * (length * breadth
                + breadth * height + length * height);
    }

    @Override
    public double volume() {
        return length * breadth * height;
    }

}
```

```java
class Circle implements Shape {
    int radius = 7;

    @Override
    public double area() {
        return Math.PI * radius * radius;
    }

    @Override
    public double volume() {
        // TODO Auto-generated method stub
        return 0;
    }

}

class Manager {
    Run | Debug
    public static void main(String[] args) {
        Shape box = new Cuboid();
        System.out.println(box.area());
        System.out.println(box.volume());

        Shape circle = new Circle();
        System.out.println(circle.area());
        System.out.println(circle.volume());
    }
}
```

Example 2) Solution

```java
interface Shape2d {
    public double area();
}

interface Shape3d {
    public double volume();
}

class Cuboid implements Shape2d, Shape3d {
    int length = 10, breadth = 20, height = 30;

    @Override
    public double area() {
        return 2 * (length * breadth
                + breadth * height + length * height);
    }

    @Override
    public double volume() {
        return length * breadth * height;
    }

}
```

```java
class Circle implements Shape2d {
    int radius = 7;

    @Override
    public double area() {
        return Math.PI * radius * radius;
    }

}

class Manager {
    Run | Debug
    public static void main(String[] args) {
        Cuboid box = new Cuboid();
        System.out.println(box.area());
        System.out.println(box.volume());

        Circle circle = new Circle();
        System.out.println(circle.area());
    }
}
```

```
public interface ParkingLot {

    void parkCar(); // Decrease empty spot count by 1
    void unparkCar(); // Increase empty spots by 1
    void getCapacity(); // Returns car capacity
    double calculateFee(Car car); // Returns the price based on number of hours
    void doPayment(Car car);
}

class Car {

}
```

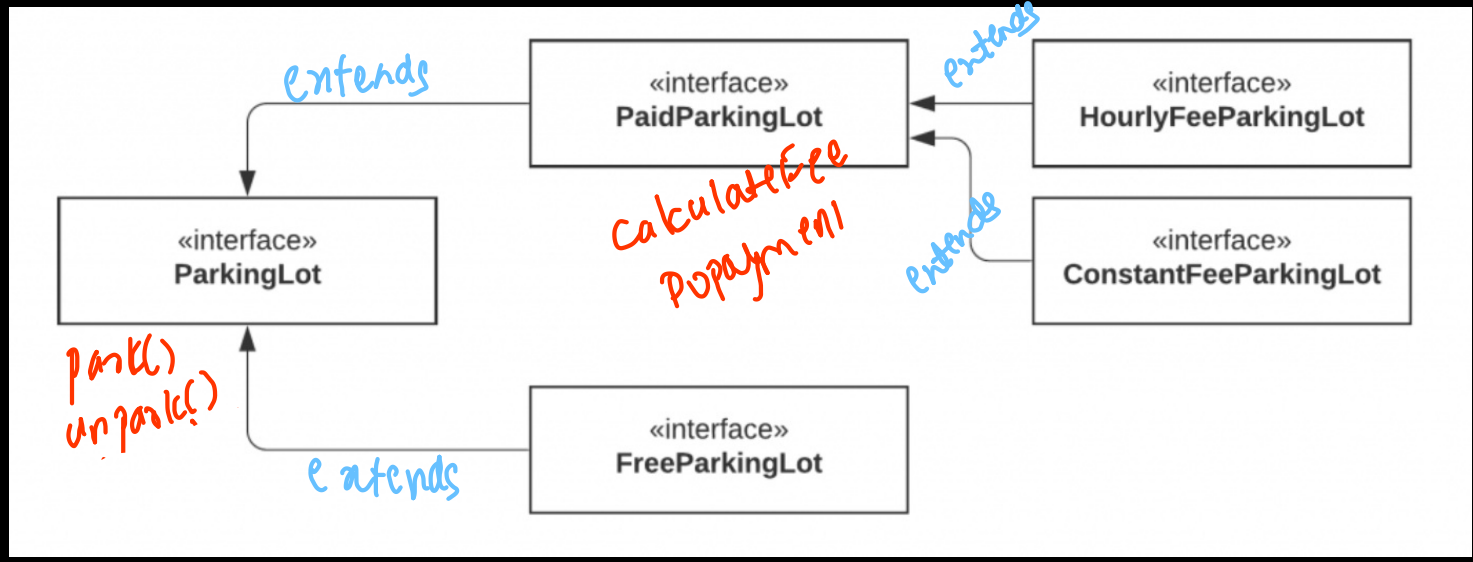
Example 3) Problem Parking lot

```
public class FreeParking implements ParkingLot {

    @Override
    public void parkCar() {

    }

    @Override
    public void unparkCar() {

    }

    @Override
    public void getCapacity() {

    }

    @Override
    public double calculateFee(Car car) {
        return 0;
    }

    @Override
    public void doPayment(Car car) {
        throw new Exception("Parking lot is free");
    }
}
```

Example 3) Solution!

Our parking lot interface was composed of 2 things: Parking related logic (park car, unpark car, get capacity) and payment related logic.

But it is too specific. Because of that, our FreeParking class was forced to implement payment-related methods that are irrelevant. Let's separate or segregate the interfaces.



We've now separated the parking lot. With this new model, we can even go further and split the **PaidParkingLot** to support different types of payment.

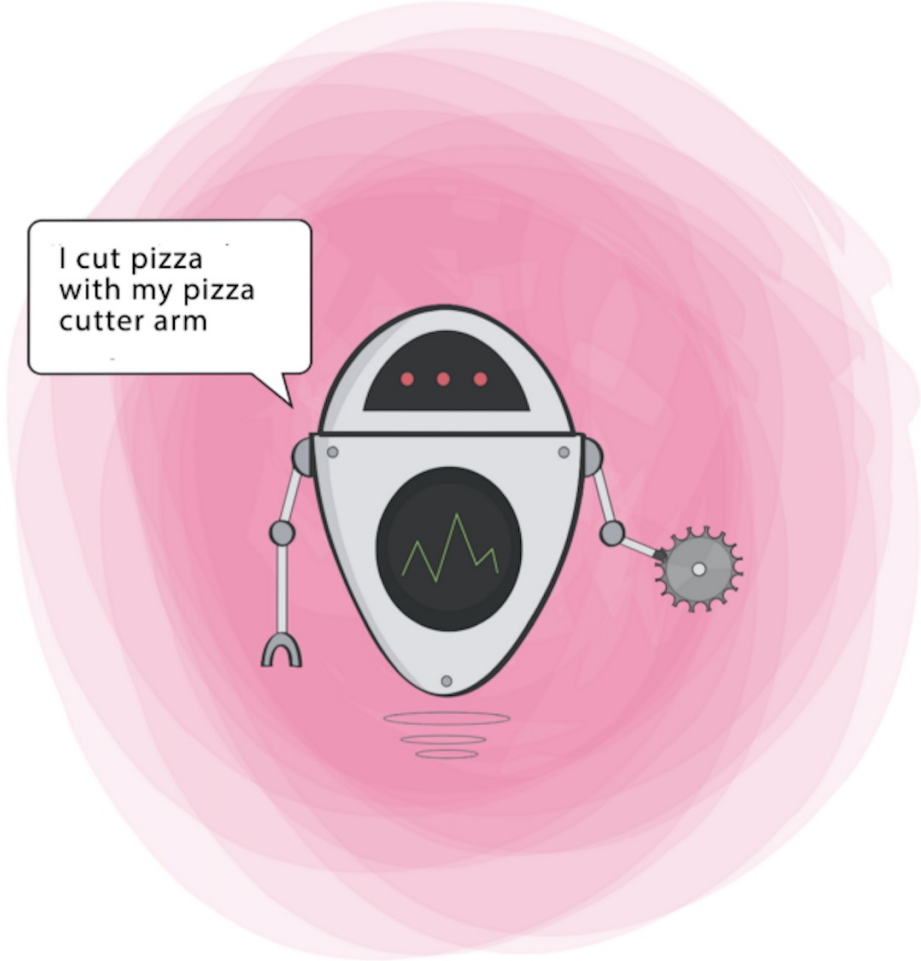
Now our model is much more flexible, extendable, and the clients do not need to implement any irrelevant logic because we provide only parking-related functionality in the parking lot interface.

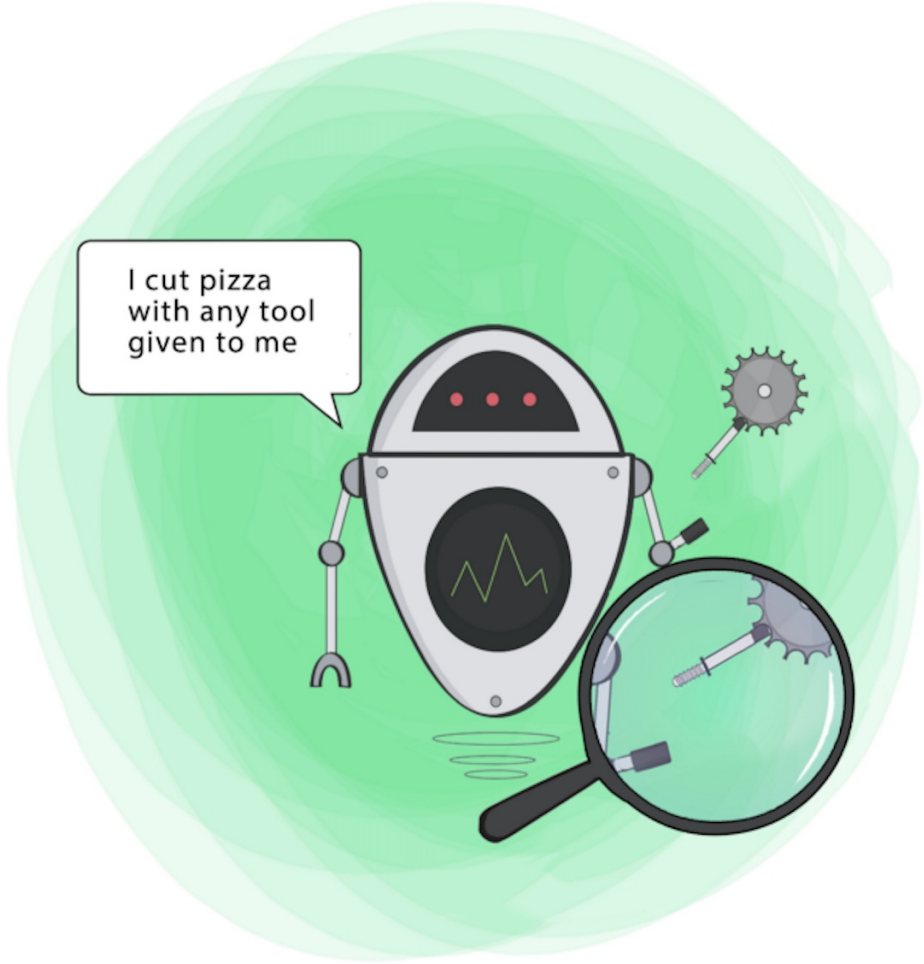
SOLID

Dependency Inversion Principle

- High level modules should not depend on low level modules.
- modules should depend upon abstractions (abstract class/interfaces) and not implementation. (concrete classes)
- Implementation should depend on abstraction but abstraction should not depend on implementation!

"Classes should upon interfaces/abstract classes and not concrete classes and methods."





Dependency Inversion

Example1 Problem

```
class Worker {
    public void work() {
        // ....working
    }
}

class Manager {
    Worker worker;

    public void setWorker(Worker w) {
        worker = w;
    }

    public void manage() {
        worker.work();
    }
}

class SuperWorker {
    public void work() {
        // .... working much more
    }
}
```

Human

Human

Human

Robot

low level class

high level module depending on concrete class (low level module)

changing manager class

modification ↑

coupling ↑

complexity ↑

New low level class

Solution

Example

```
interface IWorker {
    public void work();
}

class Worker implements IWorker {
    public void work() {
        // ....working
    }
}

class SuperWorker implements IWorker {
    public void work() {
        // .... working much more
    }
}

class Manager {
    IWorker worker;

    public void setWorker(IWorker w) {
        worker = w;
    }

    public void manage() {
        worker.work();
    }
}
```

Human

Robot

manage doesn't require changes while adding new worker type

minimized risk of breaking of manager's code ⇒ coupling ↓

no need to redo unit testing of manager

high level class depending on interface (abstraction) and not implementation !

new Robot()

new Human();

```
class SQLDatabase {
    public void create() {
    }

    public void read() {
    }

    public void update() {
    }

    public void delete() {
    }
}

class DBConnection {
    SQLDatabase db;

    DBConnection(SQLDatabase db) {
        this.db = db;
    }

    Run | Debug
    public static void main(String[] args) {
        SQLDatabase sql = new SQLDatabase();
        DBConnection dbConnection = new DBConnection(sql);
    }
}
```

tight coupling

Example2) Problem

```
class MongoDatabase {
    public void create() {
    }

    public void read() {
    }

    public void update() {
    }

    public void delete() {
    }
}
```

Example2) Solution

```
interface IDatabase {
    public void create();

    public void read();

    public void update();

    public void delete();
}

class SQLDatabase implements IDatabase {
    public void create() {
    }

    public void read() {
    }

    public void update() {
    }

    public void delete() {
    }
}
```

abstraction

concrete implementation

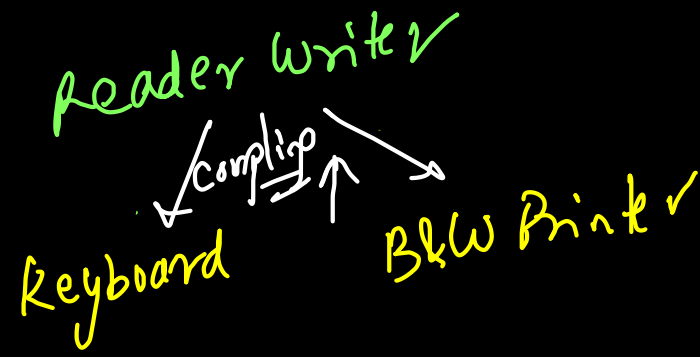
decoupling

```
class MongoDatabase implements IDatabase {
    public void create() {
    }

    public void read() {
    }

    public void update() {
    }

    public void delete() {
    }
}

class DBConnection {
    IDatabase db;

    DBConnection(IDatabase db) {
        this.db = db;
    }

    Run | Debug
    public static void main(String[] args) {
        IDatabase sql = new SQLDatabase();
        DBConnection dbConnection = new DBConnection(sql);

        IDatabase mongoose = new MongoDatabase();
        dbConnection = new DBConnection(mongoose);
    }
}
```

```java
class StandardKeyboard {
    public String read() {
        return "Keyboard Input";
    }
}

class BlackAndWhitePrinter {
    public void print(String str) {
        System.out.println("Black & White Printout : " + str);
    }
}

class ReaderWriter {

    private final StandardKeyboard reader;
    private final BlackAndWhitePrinter writer;

    public ReaderWriter() {
        reader = new StandardKeyboard();
        writer = new BlackAndWhitePrinter();
    }

    public void copy() {
        String str = reader.read();
        writer.print(str);
    }
}
```

Enample3) Problem
cleancode book

violate Dependerdency Inversion principle



Let's take the classical example of a copy module which reads characters from the keyboard and writes them to the printer device. The high level class containing the logic is the Copy class. The low level classes are KeyboardReader and PrinterWriter.

In a bad design the high level class uses directly and depends heavily on the low level classes. In such a case if we want to change the design to direct the output to a new FileWriter class we have to make changes in the Copy class. (Let's assume that it is a very complex class, with a lot of logic and really hard to test).

In order to avoid such problems we can introduce an abstraction layer between high level classes and low level classes. Since the high level modules contain the complex logic they should not depend on the low level modules so the new abstraction layer should not be created based on low level modules. Low level modules are to be created based on the abstraction layer.

```java
interface Reader {
    public String read();
}

class StandardKeyboard implements Reader {
    public String read() {
        return "Keyboard Input";
    }
}

class VoiceInput implements Reader {
    public String read() {
        return "Voice Input";
    }
}

interface Writer {
    public void print(String str);
}

class BlackAndWhitePrinter implements Writer {
    public void print(String str) {
        System.out.println("Black & White Printout : " + str);
    }
}

class ColorPrinter implements Writer {
    public void print(String str) {
        System.out.println("Colored Printout : " + str);
    }
}
```

Example 3)

Solution

```java
class ReaderWriter {

    private final Reader reader;
    private final Writer writer;

    public ReaderWriter() {
        reader = new VoiceInput();
        writer = new ColorPrinter();
    }

    public void copy() {
        String str = reader.read();
        writer.print(str);
    }
}
```