Submitted by – **Rithika**

Blackbox?: (SonarQube, Chatgpt,..)

Hides the internal working

i/p -> o/p

Ex: ATM Machine (Enter pin -> money comes out but, internal banking logic is hidden.)

Problems:

- cannot improve system design
- don't understand the real reason behind results

First-Principles Reconstruction?:

Looks at Internal Mechanism - > Rebuild the logic from scratch

Software Architecture?:

Planning , Structuring , designing how the whole system works together.

Not about coding

Code Quality?:

How good, clean, understandable, and maintainable the code is

Ex :(**Poor code** – what a, b,c represent no one knows)

```
int a=10,b=20,c=a+b;
System.out.println(c);
```

Ex :(**Good code-** Readable , understandable)

```
int marksTamil = 10;
int marksEnglish = 20;
int totalMarks = marksTamil + marksEnglish;
System.out.println(totalMarks);
```

Coupling?:

How much one part of the code depends on another part

Classes tightly connected -> tight coupling

Ex:(**Tight coupling)**

Class A depends /tied to class B

If Class B is replaced or Methods in class B is changed , then A must also be changed

D: > EI exercise > 5Days-SoftwareArchitecture > HighCouplingDemo.java

```java
class B {
    void work() {
        System.out.println("Working inside class B");
    }
}

class A {
    B obj = new B();  // direct dependency

    void doTask() {
        obj.work();
    }
}

public class HighCouplingDemo {
    public static void main(String[] args) {
        A a = new A();
        a.doTask();
    }
}
```

```
D:\EI exercise\5Days-SoftwareArchitecture>javac HighCouplingDemo.java

D:\EI exercise\5Days-SoftwareArchitecture>java HighCouplingDemo
Working inside class B
```

Classes work independently -> loose coupling

EX :(**Loose Coupling** - Uses interface to reduce dependency**)**

D: > EI exercise > 5Days-SoftwareArchitecture > J LowCouplingDemo.java

```java
interface Work {
    void work();
}

class B implements Work {
    public void work() {
        System.out.println("Working inside class B");
    }
}

class C implements Work {
    public void work() {
        System.out.println("Working inside class C");
    }
}
```

```java
class A {
    Work obj;  // depends on interface, not concrete class

    A(Work obj) {
        this.obj = obj;
    }

    void doTask() {
        obj.work();
    }
}

public class LowCouplingDemo {
    public static void main(String[] args) {
        A a1 = new A(new B()); // using B
        a1.doTask();

        A a2 = new A(new C()); // using C without changing A
        a2.doTask();
    }
}
```

```
D:\EI exercise\5Days-SoftwareArchitecture>java LowCouplingDemo
Working inside class B
Working inside class C
```

Cohesion?:

How much the parts inside a single module/class belong together and focus on doing one specific job

Low Cohesion : one class tries to do too many unrelated things.

EX:

```
class StudentManager {
   void calculateMarks() { }
   void printReport() { }
   void storeDatabase() { }
   void sendSMS() { }
}
```

High cohesion : each module/class has one clear purpose

Ex: - Each class has **one responsibility**

```
class MarksCalculator {
   void calculateMarks() { }
}

class ReportPrinter {
   void printReport() { }
}

class DatabaseStore {
   void storeDatabase() { }
}
```

```
class SmsSender {
   void sendSMS() { }
}
```

Low Cohesion: One module [Messy Module]

```
├── Marks
├── Print
├── Save DB
└── Send SMS
```

High Cohesion: each separate modules- [Marks] [Print] [DB] [SMS]

Note :

Good architecture needs**:**

- **Low Coupling** (less dependency between modules)
- **High Cohesion** (each module focused on one job)

Complexity?:

How complicated your code is to understand, test, and maintain.

Having so many conditions, branches, loops – complexity increases

Cyclomatic Complexity?:

A number that tells how many independent paths exist in your code.

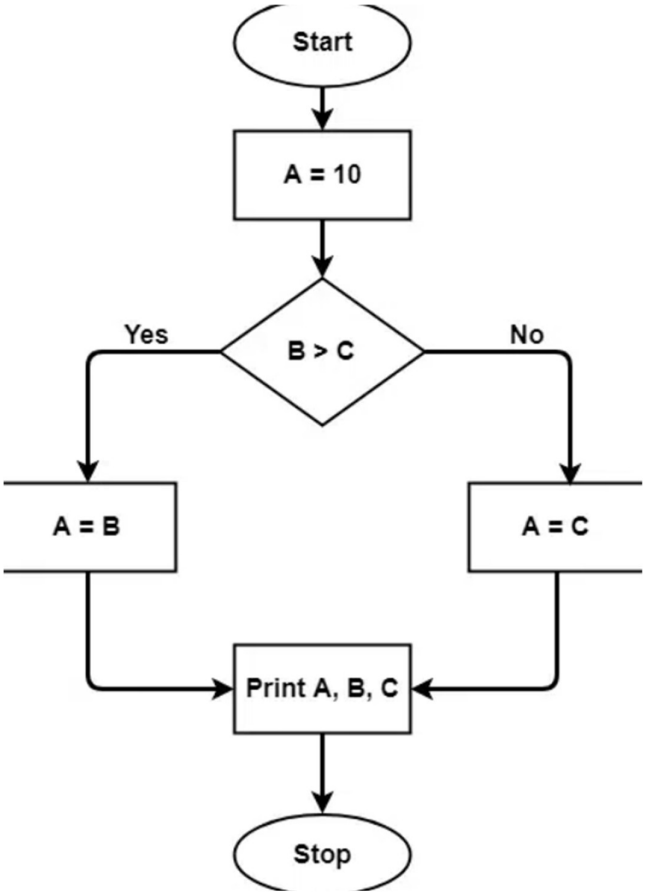
CC= Number of decision +1

M = E - N + 2P

E = the number of edges in the control flow graph
N = the number of nodes in the control flow graph
P = the number of connected components
M= Cyclomatic complexity

Ex **: E=7, N=7, P=1 = 7-7+2= 2**

```
A = 10
  IF B > C THEN
    A = B
  ELSE
    A = C
  ENDIF
Print A
Print B
Print C
```



Abstraction?

hiding details and showing only what is necessary

Virtual Method Table?:

The exact method that needs to be called is decided at runtime using a structure called the **Virtual Method Table (VTable)**.

Object Memory

| Object Header |

| VTable Pointer | ---> [Address of obj.Method]

| Fields |

---

1. Read the VTable pointer
2. Look up the correct function address
3. Jump to that function

Ex:

```
class Student {
   void study() {
      System.out.println("Student is studying general subjects");
   }
}
class ScienceStudent extends Student {
   @Override
   void study() {
      System.out.println("Science student is studying Physics and Chemistry");
   }
}
class ArtsStudent extends Student {
```

```
    @Override
    void study() {
        System.out.println("Arts student is studying History and Literature");
    }
}
public class VirtualMethodStudentDemo {
    static void startStudy(Student s) {
        s.study();
    }
    public static void main(String[] args) {
        Student s1 = new ScienceStudent();
        s1.study(); // Output: Science student is studying Physics and Chemistry
        Student s2 = new ArtsStudent();
        s2.study(); // Output: Arts student is studying History and Literature
        ScienceStudent sc = new ScienceStudent();
        sc.study(); // Output: Science student is studying Physics and Chemistry
        ArtsStudent ar = new ArtsStudent();
        ar.study(); // Output: Arts student is studying History and Literature
        System.out.println("\n--- Using method with Student reference ---");
        startStudy(s1); // Output: Science student is studying Physics and Chemistry
        startStudy(s2); // Output: Arts student is studying History and Literature
        System.out.println("\n--- Randomized student example ---");
        Student s3;
        if (Math.random() > 0.5) {
```

```
            s3 = new ScienceStudent();
        } else {
            s3 = new ArtsStudent();
        }
        s3.study(); // Could be Science or Arts depending on runtime
    }
}
```

```
D:\EI exercise\5Days-SoftwareArchitecture>java VirtualMethodStudentDemo
Science student is studying Physics and Chemistry
Arts student is studying History and Literature
Science student is studying Physics and Chemistry
Arts student is studying History and Literature

--- Using method with Student reference ---
Science student is studying Physics and Chemistry
Arts student is studying History and Literature

--- Randomized student example ---
Science student is studying Physics and Chemistry
```

Thus Abstraction have Cost - Look up VTable → then jump, Slightly slow

Compiler Optimization?(Microbenching Trap):

1.) **Devirtualization**

It converts the virtual call into a direct call, skipping the VTable lookup

Before:

VTable lookup -> call obj.Method()

After:

Direct call- > call obj.Method()

2.) **Inlining:**

Complier replace method call with actual method body

No method call , No virtual dispatch

3.) **PGO- Profile Guided Optimization:**

The runtime observes which branches and methods are actually used at runtime.

How to Force Real Virtual Dispatch?:

1.) **Reflection:** to call methods dynamically by name at runtime, so the compiler cannot optimize.
2.) **Randomized obj :** Use of random func(), to find which obj to be used at runtime so that the compiler cannot optimize.

IN Java -Randomized :

```
import java.util.Random;


class Student {
   void study() {
      int x = 0;
      x++;
   }
}


class ScienceStudent extends Student {
   @Override
   void study() {
      int x = 1;
      x++;
   }
}


class ArtsStudent extends Student {
   @Override
```

```
    void study() {
        int x = 2;
        x++;
    }
}

public class VirtualCallBenchmark {
    public static void main(String[] args) {
        final int iterations = 10_000_000;
        Random random = new Random();

        Student s1 = new ScienceStudent();
        long start1 = System.nanoTime();
        for (int i = 0; i < iterations; i++) {
            s1.study();
        }
        long end1 = System.nanoTime();
        System.out.println("Optimized virtual call (same type): " + (end1 - start1) + " ns");

        long start2 = System.nanoTime();
        for (int i = 0; i < iterations; i++) {
            Student s = random.nextBoolean() ? new ScienceStudent() : new ArtsStudent();
            s.study();
        }
```

```
        long end2 = System.nanoTime();
        System.out.println("Forced virtual call (random type): " + (end2 - start2) + " ns");
    }
}
```

```
D:\EI exercise\5Days-SoftwareArchitecture>java VirtualCallBenchmark
Optimized virtual call (same type): 4799700 ns
Forced virtual call (random type): 156212800 ns
```

IN TYPESCRIPT- Randomized obj:

```
class Student {
    study(): void {
        let x = 0;
        x++;
    }}
class ScienceStudent extends Student {
    study(): void {
        let x = 1;
        x++;
    }}
class ArtsStudent extends Student {
    study(): void {
        let x = 2;
        x++;
    }}
const iterations = 10_000_000;
let s1: Student = new ScienceStudent();
```

```
let start1 = performance.now();

for (let i = 0; i < iterations; i++) s1.study();

let end1 = performance.now();

console.log(`Optimized virtual call (same type): ${end1 - start1} ms`);

let random = () => Math.random() > 0.5;

let start2 = performance.now();

for (let i = 0; i < iterations; i++) {

    let s: Student = random() ? new ScienceStudent() : new ArtsStudent();

    s.study();

}

let end2 = performance.now();

console.log(`Forced virtual call (random type): ${end2 - start2} ms`);
```

```
D:\EI exercise\5Days-SoftwareArchitecture\ts-benchmark>npm install -g typesc
ript

added 1 package in 15s
npm notice
npm notice New major version of npm available! 10.8.2 -> 11.6.4
npm notice Changelog: https://github.com/npm/cli/releases/tag/v11.6.4
npm notice To update run: npm install -g npm@11.6.4
npm notice

D:\EI exercise\5Days-SoftwareArchitecture\ts-benchmark>tsc -v
Version 5.9.3

D:\EI exercise\5Days-SoftwareArchitecture\ts-benchmark>tsc virtualCall.ts

D:\EI exercise\5Days-SoftwareArchitecture\ts-benchmark>node virtualCall.js
--- Optimized virtual call (same type) ---
Time: 9.461199999999998 ms

--- Forced virtual call (randomized type) ---
Time: 141.84009999999998 ms
```

In Java – Reflection

```
import java.lang.reflect.Method;

class Student {

    public void study() {
```

```
        System.out.println("Student studies general subjects");
    }
}
class ScienceStudent extends Student {
    @Override
    public void study() {
        System.out.println("Science student studies Physics and Chemistry");
    }}
class ArtsStudent extends Student {
    @Override
    public void study() {
        System.out.println("Arts student studies History and Literature");
    }}
public class ReflectionDemo {
    public static void main(String[] args) throws Exception {
        Student s1 = new ScienceStudent();
        Student s2 = new ArtsStudent();


        System.out.println("--- Normal Virtual Call ---");
        Student s = s1;
        s.study();
        System.out.println("\n--- Reflection Call ---");
        s = Math.random() > 0.5 ? s1 : s2;
        Method m = s.getClass().getMethod("study");
```

```
        m.invoke(s);
    }}
```

```
D:\EI exercise\5Days-SoftwareArchitecture>java ReflectionDemo
--- Normal Virtual Call ---
Science student studies Physics and Chemistry

--- Reflection Call ---
Science student studies Physics and Chemistry
```

In TypeScript:

```
class Student {
    study(): void {
        console.log("Student studies general subjects");
    }}
class ScienceStudent extends Student {
    study(): void {
        console.log("Science student studies Physics and Chemistry");
    }}
class ArtsStudent extends Student {
    study(): void {
        console.log("Arts student studies History and Literature"); }}
const s1: Student = new ScienceStudent();
const s2: Student = new ArtsStudent();
console.log("--- Normal Virtual Call ---");
let s: Student = s1;
s.study();
console.log("\n--- Reflection Call ---");
s = Math.random() > 0.5 ? s1 : s2;
```

(s as any)["study"]();

```
D:\EI exercise\5Days-SoftwareArchitecture\ts-benchmark>tsc ReflectionDemo.ts


D:\EI exercise\5Days-SoftwareArchitecture\ts-benchmark>node ReflectionDemo
--- Normal Virtual Call ---
Science student studies Physics and Chemistry

--- Reflection Call ---
Science student studies Physics and Chemistry
```

Comparative study :

Type	**What it does**	**Optimization possibility**
Virtual call	Always ScienceStudent	JVM may inline / devirtualize → fastest
Randomized call	Random ScienceStudent or ArtsStudent	True virtual call (VTable lookup) → slower than optimized
Reflection call	Random + method via reflection	Forced runtime lookup → slowest, no compiler optimization

<u>Weighted Methods per Class (WMC)?:</u>

calculating the weight for each method.

$$\text{WMC} = \sum_{i=1}^{n} c_i$$

Metric	**High WMC**	**Low WMC**
Maintainability	Hard	Easy
Testability	Hard	Easy
Cyclomatic Complexity	High	Low
Reusability	Low	High
Example Class	Processor	CleanProcessor

Ex : IN **java**

Class **Processor**: One method with nested logic, CC is high → High WMC

Class **CleanProcessor**: Split logic → each method simple, CC low → Low WMC

Benefit: Easier to test, extend, maintain, reuse

```
// POOR: High WMC
class Processor {
    void process(String user, double amount) {
        if (user != null && !user.isEmpty()) {
            if (amount > 0) {
                System.out.println("Valid user and amount, processing...");
                if (amount < 1000) {
                    System.out.println("Small payment, fast track");
                } else {
                    System.out.println("Large payment, require approval");
                }
            } else {
                System.out.println("Invalid amount!");
            }
        } else {
            System.out.println("Invalid user!");
        }}}
// GOOD: Low WMC (Split logic into multiple methods)
class CleanProcessor {
    void process(String user, double amount) {
```

```
        if (!validateUser(user)) return;
        if (!validateAmount(amount)) return;
        handlePayment(amount);
    }
    private boolean validateUser(String user) {
        if (user == null || user.isEmpty()) {
            System.out.println("Invalid user!");
            return false;
        }
        return true;
    }
    private boolean validateAmount(double amount) {
        if (amount <= 0) {
            System.out.println("Invalid amount!");
            return false;
        }
        return true;
    }
    private void handlePayment(double amount) {
        System.out.println("Processing payment: " + amount);
        if (amount < 1000) System.out.println("Small payment, fast track");
        else System.out.println("Large payment, require approval");
    }}
public class WMCExample {
```

```
    public static void main(String[] args) {
        new Processor().process("Alice", 500);
        new CleanProcessor().process("Bob", 1500);
    }
}
```

```
D:\EI exercise\5Days-SoftwareArchitecture>javac WMCExample.java

D:\EI exercise\5Days-SoftwareArchitecture>java WMCExample
Valid user and amount, processing...
Small payment, fast track
Processing payment: 1500.0
Large payment, require approval
```

In TypeScript:

```
// POOR: High WMC
class Processor {
    process(user: string, amount: number): void {
        if (user && user.length > 0) {
            if (amount > 0) {
                console.log("Valid user and amount, processing...");
                if (amount < 1000) console.log("Small payment, fast track");
                else console.log("Large payment, require approval");
            } else {
                console.log("Invalid amount!");
            }
        } else {
            console.log("Invalid user!");
        }}}
```

```
// GOOD: Low WMC
class CleanProcessor {
    process(user: string, amount: number): void {
        if (!this.validateUser(user)) return;
        if (!this.validateAmount(amount)) return;
        this.handlePayment(amount);
    }

    private validateUser(user: string): boolean {
        if (!user || user.length === 0) {
            console.log("Invalid user!");
            return false;
        }
        return true;
    }
    private validateAmount(amount: number): boolean {
        if (amount <= 0) {
            console.log("Invalid amount!");
            return false;
        }
        return true;
    }
    private handlePayment(amount: number): void {
        console.log("Processing payment:", amount);
```

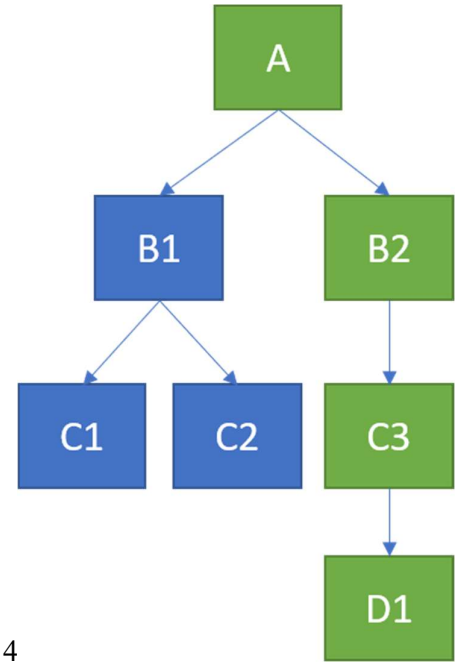
```
        if (amount < 1000) console.log("Small payment, fast track");
        else console.log("Large payment, require approval");
    }
}
new Processor().process("Alice", 500);
new CleanProcessor().process("Bob", 1500);
```

```
D:\EI exercise\5Days-SoftwareArchitecture>cd ts-benchmark

D:\EI exercise\5Days-SoftwareArchitecture\ts-benchmark>tsc WMCExample.ts

D:\EI exercise\5Days-SoftwareArchitecture\ts-benchmark>node WMCExample
Valid user and amount, processing...
Small payment, fast track
Processing payment: 1500
Large payment, require approval
```

Depth of Inheritance Tree?:

The maximum level reached by walking the dependency tree ecursively so until an element is encountered which does not Extends anything.

High DIT= Tight coupling



Here DIT = 4

In Java:

```
import java.lang.reflect.Field;


class Widget {
    protected boolean enabled = true;
    public void render() { System.out.println("Widget render (enabled=" + enabled + ")"); }
}


class Button extends Widget {
    public void render() { System.out.println("Button render (enabled=" + enabled + ")"); }
}


class ActionButton extends Button {
    public void render() { System.out.println("ActionButton render (enabled=" + enabled +
")"); }
    public void act() { System.out.println("ActionButton.act()"); }
}


class SaveButton extends ActionButton {
    public void render() {
        if (!enabled) System.out.println("SaveButton disabled");
        else System.out.println("SaveButton render (saving enabled)");
    }
}
```

```
interface Clickable {
    void click();
}

class Renderer {
    void render(String what, boolean enabled) {
        if (!enabled) System.out.println(what + " render: Disabled");
        else System.out.println(what + " render: OK");
    }
}

class PaymentHandler {
    void handlePayment(double amount) { System.out.println("Handling payment: " +
amount); }
}

class ComposeButton implements Clickable {
    private final Renderer renderer;
    private final PaymentHandler paymentHandler;
    private boolean enabled = true;

    ComposeButton(Renderer r, PaymentHandler p) {
        this.renderer = r;
        this.paymentHandler = p;
    }
```

```
    public void click() {
        if (!enabled) { System.out.println("ComposeButton: Disabled"); return; }
        renderer.render("ComposeButton", enabled);
        paymentHandler.handlePayment(42.0);
    }

    public void setEnabled(boolean e) { this.enabled = e; }
}

public class DitFragileDemo {
    public static void main(String[] args) throws Exception {
        System.out.println("=== DEEP INHERITANCE (POOR) ===");
        SaveButton save = new SaveButton();
        System.out.println("Initial call:");
        save.render();
        Field f = Widget.class.getDeclaredField("enabled");
        f.setAccessible(true);
        f.setBoolean(save, false);
        System.out.println("After base-change simulation:");
        save.render();
        System.out.println("\n=== COMPOSITION (GOOD) ===");
        Renderer r = new Renderer();
        PaymentHandler p = new PaymentHandler();
```

```
        ComposeButton cb = new ComposeButton(r, p);
        System.out.println("ComposeButton click (enabled true):");
        cb.click();
        cb.setEnabled(false);
        System.out.println("ComposeButton click (enabled false):");
        cb.click();
    }
}
```

```
D:\EI exercise\5Days-SoftwareArchitecture>java DitFragileDemo
=== DEEP INHERITANCE (POOR) ===
Initial call:
SaveButton render (saving enabled)
After changing base 'enabled' to false (simulated base change):
SaveButton disabled

=== COMPOSITION (GOOD) ===
ComposeButton click (enabled true):
ComposeButton render: OK
Handling payment: 42.0
ComposeButton click (enabled false):
ComposeButton: Disabled
```

In TypeScript:

```
class Widget {
    protected enabled: boolean = true;
    render() { console.log(`Widget render (enabled=${this.enabled})`); }
}


class Button extends Widget {
    render() { console.log(`Button render (enabled=${this.enabled})`); }
}
```

```
class ActionButton extends Button {
    render() { console.log(`ActionButton render (enabled=${this.enabled})`); }
    act() { console.log("ActionButton.act()"); }
}

class SaveButton extends ActionButton {
    render() {
        if (!this.enabled) console.log("SaveButton disabled");
        else console.log("SaveButton render (saving enabled)");
    }
}

interface IClickable { click(): void; }

class Renderer {
    render(what: string, enabled: boolean) {
        if (!enabled) console.log(`${what} render: Disabled`);
        else console.log(`${what} render: OK`);
    }
}

class PaymentHandler {
    handlePayment(amount: number) { console.log(`Handling payment: ${amount}`); }
```

```
}

class ComposeButton implements IClickable {
    private enabled = true;
    constructor(private renderer: Renderer, private paymentHandler: PaymentHandler) {}
    click(): void {
        if (!this.enabled) { console.log("ComposeButton: Disabled"); return; }
        this.renderer.render("ComposeButton", this.enabled);
        this.paymentHandler.handlePayment(99.99);
    }
    setEnabled(e: boolean) { this.enabled = e; }
}

console.log("=== DEEP INHERITANCE (POOR) ===");
const save = new SaveButton();
console.log("Initial call:");
save.render();
(save as any).enabled = false;
console.log("After base-change simulation:");
save.render();

console.log("\n=== COMPOSITION (GOOD) ===");
const renderer = new Renderer();
const payment = new PaymentHandler();
```

```
const cb = new ComposeButton(renderer, payment);

console.log("ComposeButton click (enabled true):");

cb.click();

cb.setEnabled(false);

console.log("ComposeButton click (enabled false):");

cb.click();
```

```
D:\EI exercise\5Days-SoftwareArchitecture\ts-benchmark>tsc ditFragileDemo.ts


D:\EI exercise\5Days-SoftwareArchitecture\ts-benchmark>node ditFragileDemo
=== DEEP INHERITANCE (POOR) ===
Initial call:
SaveButton render (saving enabled)
After changing base 'enabled' to false (simulated base change):
SaveButton disabled

=== COMPOSITION (GOOD) ===
ComposeButton click (enabled true):
ComposeButton render: OK
Handling payment: 99.99
ComposeButton click (enabled false):
ComposeButton: Disabled
```

Number of children: NOC

Number of immediate subclasses subordinated to the class in the class hierarchy

IN java:

Design	Meaning	Why it Matters
POOR: Concrete BaseShape with children	High NOC on concrete class	A change in BaseShape affects Circle and Square unintentionally (ripple effect — fragile)
GOOD: IShape interface with decorator + registry	High NOC on interface	Many safe extensions; no ripple effect; new shapes added without modifying existing code

```
import java.util.ArrayList;
import java.util.List;
class BaseShape {
    public void draw() {
        System.out.println("[BaseShape] set color = RED");
    }
}

class Circle extends BaseShape {
    public void draw() {
        super.draw();
        System.out.println("Circle draws");
    }
}

class Square extends BaseShape {
    public void draw() {
        super.draw();
        System.out.println("Square draws");
    }
}

interface IShape {
    void draw();
```

```
}

class GoodCircle implements IShape {
    public void draw() { System.out.println("GoodCircle draws"); }
}

class GoodSquare implements IShape {
    public void draw() { System.out.println("GoodSquare draws"); }
}

class ColoredShapeDecorator implements IShape {
    private final IShape inner;
    private final String color;
    ColoredShapeDecorator(IShape inner, String color) {
        this.inner = inner;
        this.color = color;
    }
    public void draw() {
        System.out.println("[Decorator] set color = " + color);
        inner.draw();
    }
}

class ShapeRegistry {
```

```
    private static final List<Class<? extends IShape>> implementations = new ArrayList<>();
    static void register(Class<? extends IShape> impl) { implementations.add(impl); }
    static int noc() { return implementations.size(); }
    static List<IShape> createAll() {
        List<IShape> list = new ArrayList<>();
        for (Class<? extends IShape> c : implementations) {
            try { list.add(c.getDeclaredConstructor().newInstance()); }
            catch (Exception e) {}
        }
        return list;
    }
}
public class NocDemo {
    public static void main(String[] args) {
        System.out.println("=== POOR DESIGN ===");
        List<BaseShape> poor = List.of(new Circle(), new Square());
        poor.forEach(BaseShape::draw);

        System.out.println("\nBaseShape color changed → downstream effects:");
        poor.forEach(s -> System.out.println(s.getClass().getSimpleName() + " now behaves
differently due to shared change"));

        System.out.println("\n=== GOOD DESIGN ===");
        ShapeRegistry.register(GoodCircle.class);
        ShapeRegistry.register(GoodSquare.class);
```

```
        System.out.println("NOC = " + ShapeRegistry.noc());
        ShapeRegistry.createAll().forEach(s -> new ColoredShapeDecorator(s,
"GREEN").draw());

        System.out.println("\nAdd new implementation dynamically:");
        class GoodTriangle implements IShape { public void draw() {
System.out.println("GoodTriangle draws"); } }
        ShapeRegistry.register(GoodTriangle.class);
        System.out.println("NOC = " + ShapeRegistry.noc());
        ShapeRegistry.createAll().forEach(s -> new ColoredShapeDecorator(s,
"PURPLE").draw());
    }
}
```

```
D:\EI exercise\5Days-SoftwareArchitecture>javac NocDemo.java

D:\EI exercise\5Days-SoftwareArchitecture>java NocDemo
=== POOR: High NOC on CONCRETE base class ===
[BaseShape] set color = RED  (shared impl)
Circle draws as a circle
[BaseShape] set color = RED  (shared impl)
Square draws as a square

Simulate change in BaseShape: change color default to BLUE (ripple)
[BaseShape changed] set color = BLUE  (breaking change)
Circle draws as a circle (color now BLUE)
Square draws as a square (color now BLUE)

=== GOOD: High NOC on INTERFACE (register implementations) ===
NOC for IShape (registered implementations): 2
[Decorator] set color = GREEN
GoodCircle draws; color handled externally
[Decorator] set color = GREEN
GoodSquare draws; color handled externally

Add a new implementation at runtime (plugin style) without changing interfac
e:
NOC for IShape now: 3
[Decorator] set color = PURPLE
GoodCircle draws; color handled externally
[Decorator] set color = PURPLE
GoodSquare draws; color handled externally
[Decorator] set color = PURPLE
GoodTriangle draws; color handled externally
```

In TypeScript:

```
class BaseShape {
    draw() { console.log("[BaseShape] set color = RED"); }
}
class Circle extends BaseShape {
    draw() { super.draw(); console.log("Circle draws"); }
}
class Square extends BaseShape {
    draw() { super.draw(); console.log("Square draws"); }
}

interface IShape { draw(): void; }

class GoodCircle implements IShape {
    draw() { console.log("GoodCircle draws"); }
}
class GoodSquare implements IShape {
    draw() { console.log("GoodSquare draws"); }
}

class ColoredShapeDecorator implements IShape {
    constructor(private inner: IShape, private color: string) {}
    draw() {
        console.log(`[Decorator] set color = ${this.color}`);
```

```
        this.inner.draw();
    }
}

class ShapeRegistry {
    private static impls: Array<new () => IShape> = [];
    static register(impl: new () => IShape) { this.impls.push(impl); }
    static noc(): number { return this.impls.length; }
    static createAll(): IShape[] { return this.impls.map(C => new C()); }
}

console.log("=== POOR DESIGN ===");
const poorShapes = [new Circle(), new Square()];
poorShapes.forEach(s => s.draw());

console.log("\nBaseShape changed → downstream effects:");
poorShapes.forEach(s => console.log(s.constructor.name + " affected"));

console.log("\n=== GOOD DESIGN ===");
ShapeRegistry.register(GoodCircle);
ShapeRegistry.register(GoodSquare);
console.log("NOC =", ShapeRegistry.noc());
ShapeRegistry.createAll().forEach(s => new ColoredShapeDecorator(s, "GREEN").draw());
```

```
console.log("\nAdd new implementation:");

class GoodTriangle implements IShape { draw() { console.log("GoodTriangle draws"); } }

ShapeRegistry.register(GoodTriangle);

console.log("NOC =", ShapeRegistry.noc());

ShapeRegistry.createAll().forEach(s => new ColoredShapeDecorator(s, "PURPLE").draw());
```

Design	Meaning	Why it Matters
POOR — BaseShape subclassing	High NOC on concrete class	Central logic change breaks all children
GOOD — IShape interface extensibility	High NOC on abstraction	New shapes are added safely and independently

```
D:\EI exercise\5Days-SoftwareArchitecture\ts-benchmark>tsc nocDemo.ts

D:\EI exercise\5Days-SoftwareArchitecture\ts-benchmark>node nocDemo
=== POOR: High NOC on CONCRETE base class ===
[BaseShape] set color = RED  (shared impl)
Circle draws as a circle
[BaseShape] set color = RED  (shared impl)
Square draws as a square

Simulate change in BaseShape: change color default to BLUE (ripple)
[BaseShape changed] set color = BLUE  (breaking change)
Circle draws as a circle (color now BLUE)
Square draws as a square (color now BLUE)

=== GOOD: High NOC on INTERFACE (register implementations) ===
NOC for IShape (registered implementations): 2
[Decorator] set color = GREEN
GoodCircle draws; color handled externally
[Decorator] set color = GREEN
GoodSquare draws; color handled externally

Add a new implementation at runtime (plugin style) without changing interfac
e:
NOC for IShape now: 3
[Decorator] set color = PURPLE
GoodCircle draws; color handled externally
[Decorator] set color = PURPLE
GoodSquare draws; color handled externally
[Decorator] set color = PURPLE
GoodTriangle draws; color handled externally
```

Coupling Between the Object Class?

The count of the number of other classes to which it is coupled

In java:

```
import java.util.ArrayList;
import java.util.List;


class Logger {
    void log(String msg) { System.out.println("[Logger] " + msg); }
}


class SqlDatabase {
    void save(String payload) { System.out.println("[SqlDatabase] saved: " + payload); }
}


class LegacyDb {
    void persist(String payload) { System.out.println("[LegacyDb] persisted: " + payload); }
}


class LegacyDbAdapter extends SqlDatabase {
    private final LegacyDb legacy = new LegacyDb();
    @Override
    void save(String payload) { legacy.persist(payload); }
}
```

```
class Order {
    final String id;
    final double amount;
    Order(String id, double amount) { this.id = id; this.amount = amount; }
    public String toString() { return id + ":" + amount; }
}

class OrderService {
    void checkout(Order o) {
        Logger logger = new Logger();
        SqlDatabase db = new SqlDatabase();
        logger.log("Processing " + o);
        db.save(o.toString());
        logger.log("Completed " + o);
    }
}

interface ILogger { void log(String msg); }
interface IDatabase { void save(String payload); }

class CleanOrderService {
    private final ILogger logger;
    private final IDatabase db;
    CleanOrderService(ILogger logger, IDatabase db) { this.logger = logger; this.db = db; }
```

```
    void checkout(Order o) {
        logger.log("Processing " + o);
        db.save(o.toString());
        logger.log("Completed " + o);
    }
}

class MockLogger implements ILogger { public void log(String msg) {
System.out.println("[MockLogger] " + msg); } }
class MockDatabase implements IDatabase { public void save(String payload) {
System.out.println("[MockDatabase] saved: " + payload); } }

interface EventListener { void on(Object event); }

class EventBus {
    private final List<EventListener> listeners = new ArrayList<>();
    void subscribe(EventListener l) { listeners.add(l); }
    void publish(Object e) { for (EventListener l : listeners) l.on(e); }
}

class EventOrderService {
    private final EventBus bus;
    EventOrderService(EventBus bus) { this.bus = bus; }
    void checkout(Order o) { bus.publish(o); }
}
```

```
class OrderHandler implements EventListener {
    private final ILogger logger;
    private final IDatabase db;
    OrderHandler(ILogger logger, IDatabase db) { this.logger = logger; this.db = db; }
    public void on(Object event) {
        if (!(event instanceof Order)) return;
        Order o = (Order) event;
        logger.log("Event handling " + o);
        db.save(o.toString());
    }
}
public class OrderCboDemo {
    public static void main(String[] args) {
        System.out.println("=== POOR: High CBO (tight coupling) ===");
        OrderService poor = new OrderService();
        poor.checkout(new Order("o-1", 99.9));
        System.out.println("\n=== GOOD: Low CBO (DI via interfaces) ===");
        CleanOrderService clean = new CleanOrderService(new MockLogger(), new
MockDatabase());
        clean.checkout(new Order("o-2", 199.9));
        System.out.println("\n=== GOOD: Adapter for legacy DB (still low CBO) ===");
        CleanOrderService adapted = new CleanOrderService(new Logger()::log, new
LegacyDbAdapter()::save);
        adapted.checkout(new Order("o-3", 299.9));
```

```
        System.out.println("\n=== GOOD: Event-driven decoupling ===");
        EventBus bus = new EventBus();
        OrderHandler handler = new OrderHandler(new Logger()::log, new MockDatabase());
        bus.subscribe(handler);
        EventOrderService evtService = new EventOrderService(bus);
        evtService.checkout(new Order("o-4", 49.9));
    }
}
```

```
D:\EI exercise\5Days-SoftwareArchitecture>java OrderCboDemo
=== POOR: High CBO (tight coupling) ===
[Logger] Processing o-1:99.9
[SqlDatabase] saved: o-1:99.9
[Logger] Completed o-1:99.9

=== GOOD: Low CBO (DI via interfaces) ===
[MockLogger] Processing o-2:199.9
[MockDatabase] saved: o-2:199.9
[MockLogger] Completed o-2:199.9

=== GOOD: Adapter for legacy DB (still low CBO) ===
[Logger] Processing o-3:299.9
[LegacyDb] persisted: o-3:299.9
[Logger] Completed o-3:299.9

=== GOOD: Event-driven decoupling ===
[Logger] Event handling o-4:49.9
[MockDatabase] saved: o-4:49.9
```

In TypeScript:

```
class Logger {
    log(msg: string) { console.log("[Logger] " + msg); }
}
class SqlDatabase {
    save(payload: string) { console.log("[SqlDatabase] saved: " + payload); }
}
```

```
class LegacyDb {
    persist(payload: string) { console.log("[LegacyDb] persisted: " + payload); }
}
class LegacyDbAdapter extends SqlDatabase {
    private legacy = new LegacyDb();
    save(payload: string) { this.legacy.persist(payload); }
}
type Order = { id: string, amount: number };

class OrderService {
    checkout(o: Order) {
        const logger = new Logger();
        const db = new SqlDatabase();
        logger.log("Processing " + JSON.stringify(o));
        db.save(JSON.stringify(o));
        logger.log("Completed " + JSON.stringify(o));
    }
}

interface ILogger { log(msg: string): void; }
interface IDatabase { save(payload: string): void; }

class CleanOrderService {
    constructor(private logger: ILogger, private db: IDatabase) {}
```

```
    checkout(o: Order) {
        this.logger.log("Processing " + JSON.stringify(o));
        this.db.save(JSON.stringify(o));
        this.logger.log("Completed " + JSON.stringify(o));
    }
}
class MockLogger implements ILogger { log(msg: string) { console.log("[MockLogger] " +
msg); } }
class MockDatabase implements IDatabase { save(payload: string) {
console.log("[MockDatabase] saved: " + payload); } }
class EventBus {
    private listeners: Array<(e: any) => void> = [];
    subscribe(fn: (e: any) => void) { this.listeners.push(fn); }
    publish(e: any) { this.listeners.forEach(fn => fn(e)); }
}
class EventOrderService {
    constructor(private bus: EventBus) {}
    checkout(o: Order) { this.bus.publish(o); }
}
function orderHandlerFactory(logger: ILogger, db: IDatabase) {
    return (e: any) => {
        if (!e || !e.id) return;
        logger.log("Event handling " + JSON.stringify(e));
        db.save(JSON.stringify(e));
    };
```

```
}

console.log("=== POOR: High CBO (tight coupling) ===");

new OrderService().checkout({ id: "o-1", amount: 99.9 });

console.log("\n=== GOOD: Low CBO (DI via interfaces) ===");

new CleanOrderService(new MockLogger(), new MockDatabase()).checkout({ id: "o-2",
amount: 199.9 });

console.log("\n=== GOOD: Adapter for legacy DB ===");

new CleanOrderService(new Logger(), new LegacyDbAdapter()).checkout({ id: "o-3",
amount: 299.9 });

console.log("\n=== GOOD: Event-driven decoupling ===");

const bus = new EventBus();

bus.subscribe(orderHandlerFactory(new Logger(), new MockDatabase()));

const evt = new EventOrderService(bus);

evt.checkout({ id: "o-4", amount: 49.9 });
```

```
D:\EI exercise\5Days-SoftwareArchitecture\ts-benchmark>tsc orderCboDemo.ts

D:\EI exercise\5Days-SoftwareArchitecture\ts-benchmark>node orderCboDemo
=== POOR: High CBO (tight coupling) ===
[Logger] Processing {"id":"o-1","amount":99.9}
[SqlDatabase] saved: {"id":"o-1","amount":99.9}
[Logger] Completed {"id":"o-1","amount":99.9}

=== GOOD: Low CBO (DI via interfaces) ===
[MockLogger] Processing {"id":"o-2","amount":199.9}
[MockDatabase] saved: {"id":"o-2","amount":199.9}
[MockLogger] Completed {"id":"o-2","amount":199.9}

=== GOOD: Adapter for legacy DB ===
[Logger] Processing {"id":"o-3","amount":299.9}
[LegacyDb] persisted: {"id":"o-3","amount":299.9}
[Logger] Completed {"id":"o-3","amount":299.9}

=== GOOD: Event-driven decoupling ===
[Logger] Event handling {"id":"o-4","amount":49.9}
[MockDatabase] saved: {"id":"o-4","amount":49.9}
```

Response For a Class (RFC)?

|RS| where RS= Response set fr the class

RS= { M} U {Ri}

{Ri}-> set of method called by method i

{M} -> set of all methods in the class

In java:

```
import java.util.Random;

class AuthService {

boolean checkCredentials(String user) { return true; }

}

class Database {

void connect() { System.out.println("[DB] connect"); }

void save(String payload) { System.out.println("[DB] save: " + payload); }

void close() { System.out.println("[DB] close"); }

}

class EmailService {

void sendReceipt(String to, String body) { System.out.println("[Email] to=" + to); }

}

class Logger {

void info(String m) { System.out.println("[Log] " + m); }

void error(String m) { System.out.println("[Log-Err] " + m); }

}

class Order {

final String id; final double amount;
```

```
Order(String id, double amount){ this.id = id; this.amount = amount; }
public String toString(){ return id + ":" + amount; }
}
class OrderProcessor { // POOR: chatty object (high RFC)
private final AuthService auth = new AuthService();
private final Database db = new Database();
private final EmailService email = new EmailService();
private final Logger logger = new Logger();
void process(Order o, String user) {
logger.info("Start " + o);
if (!auth.checkCredentials(user)) { logger.error("auth fail"); return; }
db.connect();
db.save(o.toString());
email.sendReceipt(user, "Thanks for order " + o.id);
logger.info("Done " + o);
db.close();
}
// illustrative RFC count (not computed via analysis)
static int illustrativeRFC() {
int M = 1; // methods owned (process)
int R = 6; // external methods called: auth.checkCredentials, db.connect, db.save,
email.sendReceipt, logger.info, db.close
return M + R;
}
}
```

```
class TransactionFacade { // Facade that groups many calls
private final AuthService auth;
private final Database db;
private final EmailService email;
private final Logger logger;
TransactionFacade(AuthService auth, Database db, EmailService email, Logger logger) {
this.auth = auth; this.db = db; this.email = email; this.logger = logger;
}
boolean handleTransaction(Order o, String user) {
logger.info("Facade start " + o);
if (!auth.checkCredentials(user)) { logger.error("auth fail"); return false; }
db.connect();
db.save(o.toString());
email.sendReceipt(user, "Thanks for order " + o.id);
logger.info("Facade done " + o);
db.close();
return true;
}
}
class CleanProcessor { // GOOD: low RFC because it delegates to a single facade call
private final TransactionFacade facade;
CleanProcessor(TransactionFacade facade){ this.facade = facade; }
void process(Order o, String user){
facade.handleTransaction(o, user);
```

```
}
static int illustrativeRFC(){
int M = 1; // process
int R = 1; // facade.handleTransaction
return M + R;
}
}
public class RfcDemo {
public static void main(String[] args){
Order o1 = new Order("o-101", 49.99);
Order o2 = new Order("o-202", 149.50);
System.out.println("=== POOR: High RFC (chatty) ===");
OrderProcessor p = new OrderProcessor();
p.process(o1, "alice@example.com");
System.out.println("OrderProcessor RFC (illustrative) = " +
OrderProcessor.illustrativeRFC());
System.out.println("\n=== GOOD: Low RFC (facade) ===");
TransactionFacade facade = new TransactionFacade(new AuthService(), new Database(),
new EmailService(), new Logger());
CleanProcessor cp = new CleanProcessor(facade);
cp.process(o2, "bob@example.com");
System.out.println("CleanProcessor RFC (illustrative) = " +
CleanProcessor.illustrativeRFC());
System.out.println("\n=== Creative: Run many mixed processors to show behavior ===");
Random rnd = new Random();
```

```
for (int i=0;i<3;i++){

Order o = new Order("batch-"+i, rnd.nextDouble()*200);

if (rnd.nextBoolean()) p.process(o, "user@x");

else cp.process(o, "user@y");

} } }
```

```
D:\EI exercise\5Days-SoftwareArchitecture>java RfcDemo
=== POOR: High RFC (chatty) ===
[Log] Start o-101:49.99
[DB] connect
[DB] save: o-101:49.99
[Email] to=alice@example.com
[Log] Done o-101:49.99
[DB] close
OrderProcessor RFC (illustrative) = 7

=== GOOD: Low RFC (facade) ===
[Log] Facade start o-202:149.5
[DB] connect
[DB] save: o-202:149.5
[Email] to=bob@example.com
[Log] Facade done o-202:149.5
[DB] close
CleanProcessor RFC (illustrative) = 2

=== Creative: Run many mixed processors to show behavior ===
[Log] Facade start batch-0:121.48022293438186
[DB] connect
[DB] save: batch-0:121.48022293438186
[Email] to=user@y
[Log] Facade done batch-0:121.48022293438186
[DB] close
[Log] Facade start batch-1:138.34714178855597
[DB] connect
[DB] save: batch-1:138.34714178855597
[Email] to=user@y
[Log] Facade done batch-1:138.34714178855597
[DB] close
[Log] Start batch-2:107.08636722873514
[DB] connect
[DB] save: batch-2:107.08636722873514
[Email] to=user@x
[Log] Done batch-2:107.08636722873514
[DB] close
```

In typescript:

```
class AuthService {
    checkCredentials(user: string): boolean { return true; }
}
class Database {
    connect() { console.log("[DB] connect"); }
    save(payload: string) { console.log("[DB] save:", payload); }
    close() { console.log("[DB] close"); }
}
class EmailService {
    sendReceipt(to: string, body: string) { console.log("[Email] to=", to); }
}
class Logger {
    info(m: string) { console.log("[Log]", m); }
    error(m: string) { console.log("[Log-Err]", m); }
}
type Order = { id: string; amount: number; };

class OrderProcessor {
    private auth = new AuthService();
    private db = new Database();
    private email = new EmailService();
    private logger = new Logger();
```

```
    process(o: Order, user: string) {
        this.logger.info("Start " + JSON.stringify(o));
        if (!this.auth.checkCredentials(user)) { this.logger.error("auth fail"); return; }
        this.db.connect();
        this.db.save(JSON.stringify(o));
        this.email.sendReceipt(user, "Thanks " + o.id);
        this.logger.info("Done " + JSON.stringify(o));
        this.db.close();
    }

    static illustrativeRFC(): number {
        const M = 1;
        const R = 6;
        return M + R;
    }
}

class TransactionFacade {
    constructor(private auth: AuthService, private db: Database, private email: EmailService,
private logger: Logger) {}
    handleTransaction(o: Order, user: string): boolean {
        this.logger.info("Facade start " + JSON.stringify(o));
        if (!this.auth.checkCredentials(user)) { this.logger.error("auth fail"); return false; }
        this.db.connect();
        this.db.save(JSON.stringify(o));
```

```
        this.email.sendReceipt(user, "Thanks " + o.id);
        this.logger.info("Facade done " + JSON.stringify(o));
        this.db.close();
        return true;
    }
}

class CleanProcessor {
    constructor(private facade: TransactionFacade) {}
    process(o: Order, user: string) { this.facade.handleTransaction(o, user); }
    static illustrativeRFC(): number { return 1 + 1; }
}

console.log("=== POOR: High RFC (chatty) ===");
const poor = new OrderProcessor();
poor.process({ id: "o-11", amount: 9.99 }, "alice@x.com");
console.log("OrderProcessor RFC (illustrative) =", OrderProcessor.illustrativeRFC());

console.log("\n=== GOOD: Low RFC (facade) ===");
const facade = new TransactionFacade(new AuthService(), new Database(), new
EmailService(), new Logger());
const clean = new CleanProcessor(facade);
clean.process({ id: "o-22", amount: 29.99 }, "bob@y.com");
console.log("CleanProcessor RFC (illustrative) =", CleanProcessor.illustrativeRFC());
```

```
console.log("\n=== Creative: batch run ===");
for (let i=0;i<3;i++){
  const o = { id: `batch-${i}`, amount: Math.random()*300 };
  if (Math.random() > 0.5) poor.process(o, "u@x");
  else clean.process(o, "u@y");
}
```

```
D:\EI exercise\5Days-SoftwareArchitecture\ts-benchmark>tsc rfcDemo.ts

D:\EI exercise\5Days-SoftwareArchitecture\ts-benchmark>node rfcDemo
=== POOR: High RFC (chatty) ===
[Log] Start {"id":"o-11","amount":9.99}
[DB] connect
[DB] save: {"id":"o-11","amount":9.99}
[Email] to= alice@x.com
[Log] Done {"id":"o-11","amount":9.99}
[DB] close
OrderProcessor RFC (illustrative) = 7

=== GOOD: Low RFC (facade) ===
[Log] Facade start {"id":"o-22","amount":29.99}
[DB] connect
[DB] save: {"id":"o-22","amount":29.99}
[Email] to= bob@y.com
[Log] Facade done {"id":"o-22","amount":29.99}
[DB] close
CleanProcessor RFC (illustrative) = 2

=== Creative: batch run ===
[Log] Facade start {"id":"batch-0","amount":59.457542281287104}
[DB] connect
[DB] save: {"id":"batch-0","amount":59.457542281287104}
[Email] to= u@y
[Log] Facade done {"id":"batch-0","amount":59.457542281287104}
[DB] close
[Log] Facade start {"id":"batch-1","amount":108.3302840796811}
[DB] connect
[DB] save: {"id":"batch-1","amount":108.3302840796811}
[Email] to= u@y
[Log] Facade done {"id":"batch-1","amount":108.3302840796811}
[DB] close
[Log] Facade start {"id":"batch-2","amount":76.03352269113508}
[DB] connect
[DB] save: {"id":"batch-2","amount":76.03352269113508}
[Email] to= u@y
[Log] Facade done {"id":"batch-2","amount":76.03352269113508}
[DB] close
```

Lack of Cohesion of Methods (LCOM)?

LCOM4 counts the connected components. A score greater than 1 implies the class is trying to be two different things and should be split

In java

```
import java.util.*;


class LcomCalculator {
   static int lcom4(List<MethodAccess> methods) {
      int n = methods.size();
      boolean[][] adj = new boolean[n][n];
      for (int i = 0; i < n; i++) {
         for (int j = i+1; j < n; j++) {
            Set<String> a = methods.get(i).fields;
            Set<String> b = methods.get(j).fields;
            boolean intersect = false;
            for (String f : a) {
               if (b.contains(f)) { intersect = true; break; }
            }
            adj[i][j] = adj[j][i] = intersect;
         }
      }
      boolean[] seen = new boolean[n];
      int components = 0;
      for (int i = 0; i < n; i++) {
         if (!seen[i]) {
```

```
            components++;
            Deque<Integer> stack = new ArrayDeque<>();
            stack.push(i);
            seen[i] = true;
            while (!stack.isEmpty()) {
                int u = stack.pop();
                for (int v = 0; v < n; v++) {
                    if (!seen[v] && adj[u][v]) {
                        seen[v] = true;
                        stack.push(v);
                    }
                }
            }
        }
    }
    return components;
}

static class MethodAccess {
    final String name;
    final Set<String> fields;
    MethodAccess(String name, String... fields) {
        this.name = name;
        this.fields = new HashSet<>(Arrays.asList(fields));
```

```
        }
    }
}

class PoorUserManager {
    String name = "rithika";
    String email = "rith@gmail";
    String dbConfig = "db";
    String auditLog = "log";
    void printName() { System.out.println(name); }
    void sendWelcomeEmail() { System.out.println("email->" + email); }
    void connect() { System.out.println("db->" + dbConfig); }
    void persistUser() { System.out.println("persisting " + name); }
    void writeAudit() { System.out.println("audit:" + auditLog); }
}

class User {
    String name = "rithika";
    void printName() { System.out.println(name); }
}

class DbConnector {
    String dbConfig = "db";
    void connect() { System.out.println("db->" + dbConfig); }
```

```
    void persistUser(String name) { System.out.println("persisting " + name); }
}

class Notifier {
    String email = "rith@gmail";
    void sendWelcomeEmail(String to) { System.out.println("email->" + to); }
}

class Auditor {
    String auditLog = "log";
    void writeAudit(String entry) { System.out.println("audit:" + entry); }
}

public class LcomDemo {
    public static void main(String[] args) {
        List<LcomCalculator.MethodAccess> poorMethods = Arrays.asList(
            new LcomCalculator.MethodAccess("printName", "name"),
            new LcomCalculator.MethodAccess("sendWelcomeEmail", "email"),
            new LcomCalculator.MethodAccess("connect", "dbConfig"),
            new LcomCalculator.MethodAccess("persistUser", "dbConfig","name"),
            new LcomCalculator.MethodAccess("writeAudit", "auditLog")
        );

        int poorLcom = LcomCalculator.lcom4(poorMethods);
```

```
        System.out.println("POOR class methods components (LCOM4) = " + poorLcom);


        List<LcomCalculator.MethodAccess> userMethods = Arrays.asList(
            new LcomCalculator.MethodAccess("printName", "name")
        );
        List<LcomCalculator.MethodAccess> dbMethods = Arrays.asList(
            new LcomCalculator.MethodAccess("connect", "dbConfig"),
            new LcomCalculator.MethodAccess("persistUser", "dbConfig","name")
        );
        List<LcomCalculator.MethodAccess> notifierMethods = Arrays.asList(
            new LcomCalculator.MethodAccess("sendWelcomeEmail", "email")
        );
        List<LcomCalculator.MethodAccess> auditorMethods = Arrays.asList(
            new LcomCalculator.MethodAccess("writeAudit", "auditLog")
        );


        System.out.println("GOOD class User LCOM4 = " +
LcomCalculator.lcom4(userMethods));
        System.out.println("GOOD class DbConnector LCOM4 = " +
LcomCalculator.lcom4(dbMethods));
        System.out.println("GOOD class Notifier LCOM4 = " +
LcomCalculator.lcom4(notifierMethods));
        System.out.println("GOOD class Auditor LCOM4 = " +
LcomCalculator.lcom4(auditorMethods));
```

```
        System.out.println("\nRefactored run:");
        User u = new User(); u.name = "rithika"; u.printName();
        DbConnector db = new DbConnector(); db.dbConfig = "db://x"; db.connect();
db.persistUser("rithika");
        Notifier n = new Notifier(); n.email = "rith@gmail";
n.sendWelcomeEmail("rith@gmail");
        Auditor a = new Auditor(); a.auditLog = "log01"; a.writeAudit("created user rithika");
    }
}
```

```
D:\EI exercise\5Days-SoftwareArchitecture\ts-benchmark>cd ..

D:\EI exercise\5Days-SoftwareArchitecture>javac LcomDemo.java

D:\EI exercise\5Days-SoftwareArchitecture>java LcomDemo
POOR class methods components (LCOM4) = 3
GOOD class User LCOM4 = 1
GOOD class DbConnector LCOM4 = 1
GOOD class Notifier LCOM4 = 1
GOOD class Auditor LCOM4 = 1

Refactored run:
rithika
db->db://x
persisting rithika
email->rith@gmail
audit:created user rithika
```

In typescript:

```
class LcomCalculator {
    static lcom4(methods: {name: string; fields: string[]}[]): number {
        var n = methods.length;
        var adj: boolean[][] = [];
        for (var i = 0; i < n; i++) {
            adj[i] = [];
```

```
    for (var j = 0; j < n; j++) adj[i][j] = false;
}
for (var i = 0; i < n; i++) {
    for (var j = i + 1; j < n; j++) {
        var aFields = methods[i].fields;
        var bFields = methods[j].fields;
        var intersect = false;
        for (var ia = 0; ia < aFields.length; ia++) {
            var f = aFields[ia];
            for (var jb = 0; jb < bFields.length; jb++) {
                if (bFields[jb] === f) { intersect = true; break; }
            }
            if (intersect) break;
        }
        adj[i][j] = adj[j][i] = intersect;
    }
}
var seen: boolean[] = [];
for (var i = 0; i < n; i++) seen[i] = false;
var components = 0;
for (var i = 0; i < n; i++) {
    if (!seen[i]) {
        components++;
        var stack = [i];
```

```
            seen[i] = true;
            while (stack.length) {
                var u = stack.pop() as number;
                for (var v = 0; v < n; v++) {
                    if (!seen[v] && adj[u][v]) { seen[v] = true; stack.push(v); }
                }
            }
        }
    }
    return components;
  }
}

class PoorUserManager {
  name = "rithika";
  email = "rith@gmail";
  dbConfig = "db";
  auditLog = "log";
  printName() { console.log(this.name); }
  sendWelcomeEmail() { console.log("email->", this.email); }
  connect() { console.log("db->", this.dbConfig); }
  persistUser() { console.log("persist", this.name); }
  writeAudit() { console.log("audit", this.auditLog); }
}
```

```
class User {
    name = "rithika";
    printName() { console.log(this.name); }
}

class DbConnector {
    dbConfig = "db";
    connect() { console.log("db->", this.dbConfig); }
    persistUser(name: string) { console.log("persist", name); }
}

class Notifier {
    email = "rith@gmail";
    sendWelcomeEmail(to: string) { console.log("email->", to); }
}

class Auditor {
    auditLog = "log";
    writeAudit(entry: string) { console.log("audit", entry); }
}

var poorMethods = [
    {name: "printName", fields:["name"]},
```

```
    {name: "sendWelcomeEmail", fields:["email"]},
    {name: "connect", fields:["dbConfig"]},
    {name: "persistUser", fields:["dbConfig","name"]},
    {name: "writeAudit", fields:["auditLog"]}
];

console.log("POOR class methods components (LCOM4) =",
LcomCalculator.lcom4(poorMethods));

var userMethods = [{name:"printName", fields:["name"]}];
var dbMethods = [
    {name:"connect", fields:["dbConfig"]},
    {name:"persistUser", fields:["dbConfig","name"]}
];
var notifierMethods = [{name:"sendWelcomeEmail", fields:["email"]}];
var auditorMethods = [{name:"writeAudit", fields:["auditLog"]}];

console.log("GOOD class User LCOM4 =", LcomCalculator.lcom4(userMethods));
console.log("GOOD class DbConnector LCOM4 =", LcomCalculator.lcom4(dbMethods));
console.log("GOOD class Notifier LCOM4 =", LcomCalculator.lcom4(notifierMethods));
console.log("GOOD class Auditor LCOM4 =", LcomCalculator.lcom4(auditorMethods));

console.log("\nRefactored run:");
var u = new User(); u.name = "rithika"; u.printName();
var db = new DbConnector(); db.dbConfig = "db://x"; db.connect(); db.persistUser("rithika");
```

var n = new Notifier(); n.email = "rith@gmail"; n.sendWelcomeEmail("rith@gmail");

var a = new Auditor(); a.auditLog = "log01"; a.writeAudit("created user rithika");

```
D:\EI exercise\5Days-SoftwareArchitecture\ts-benchmark>node lcomDemo
POOR class methods components (LCOM4) = 3
GOOD class User LCOM4 = 1
GOOD class DbConnector LCOM4 = 1
GOOD class Notifier LCOM4 = 1
GOOD class Auditor LCOM4 = 1

Refactored run:
rithika
db-> db://x
persist rithika
email-> rith@gmail
audit created user rithika
```