**SOLID Principle কি?**

SOLID হল পাঁচটি ডিজাইন প্রিন্সিপলের সমষ্টি যা অবজেক্ট-ওরিয়েন্টেড ডিজাইনকে আরও ভালো, মেইনটেনেবল ও স্কেলেবল করে। এগুলো ব্যবহার করলে কোড REUSABLE হয়, পরিবর্তন করা সহজ হয় এবং ERRORS কমে।

SOLID মানে:

* **S** – **Single Responsibility Principle (SRP)**
* **O** – **Open/Closed Principle (OCP)**
* **L** – **Liskov Substitution Principle (LSP)**
* **I** – **Interface Segregation Principle (ISP)**
* **D** – **Dependency Inversion Principle (DIP)**

এখন এগুলোকে বিস্তারিত বুঝি।

**1. Single Responsibility Principle (SRP)**

**একটি ক্লাসের মাত্র একটিই কাজ থাকা উচিত।**

যদি একটি ক্লাস একাধিক কাজ করে, তাহলে পরিবর্তন করার সময় একাধিক কারণে পরিবর্তন করতে হতে পারে, যা মেইনটেনেন্স কঠিন করে তোলে।

🔹 **ভুল উদাহরণ (Bad Example):**

class Report {

public function generateReport() {

// রিপোর্ট তৈরি করা

}

public function saveToFile() {

// রিপোর্ট ফাইলে সংরক্ষণ করা

}

}

এখানে **Report** ক্লাস দুটি কাজ করছে –

1. রিপোর্ট তৈরি করছে।
2. রিপোর্ট ফাইলে সংরক্ষণ করছে।

👉 **ভালো উদাহরণ (Good Example):**

class ReportGenerator {

public function generate() {

// রিপোর্ট তৈরি করা

}

}

class ReportSaver {

public function saveToFile($report) {

// রিপোর্ট ফাইলে সংরক্ষণ করা

}

}

এখানে **ReportGenerator** শুধু রিপোর্ট তৈরি করছে, আর **ReportSaver** ফাইলে সংরক্ষণ করছে।

**2. Open/Closed Principle (OCP)**

**একটি ক্লাস পরিবর্তনের জন্য বন্ধ, কিন্তু এক্সটেনশন (বর্ধিতকরণ) এর জন্য খোলা থাকবে।**

যদি নতুন ফিচার যোগ করতে হয়, তাহলে পুরাতন কোড পরিবর্তন না করে নতুন কোড লিখে এক্সটেনশন করা উচিত।

🔹 **ভুল উদাহরণ (Bad Example):**

class AreaCalculator {

public function calculate($shape) {

if ($shape instanceof Circle) {

return pi() \* pow($shape->radius, 2);

} elseif ($shape instanceof Rectangle) {

return $shape->width \* $shape->height;

}

}

}

এখানে নতুন **Triangle** যোগ করতে হলে মূল ক্লাস পরিবর্তন করতে হবে।

👉 **ভালো উদাহরণ (Good Example):**

interface Shape {

public function area();

}

class Circle implements Shape {

public function \_\_construct(private $radius) {}

public function area() {

return pi() \* pow($this->radius, 2);

}

}

class Rectangle implements Shape {

public function \_\_construct(private $width, private $height) {}

public function area() {

return $this->width \* $this->height;

}

}

class AreaCalculator {

public function calculate(Shape $shape) {

return $shape->area();

}

}

এখন নতুন **Triangle** যোগ করলে **AreaCalculator** পরিবর্তন করতে হবে না, শুধু নতুন ক্লাস লিখলেই হবে।

**3. Liskov Substitution Principle (LSP)**

**একটি সাবক্লাসকে প্যারেন্ট ক্লাস দিয়ে প্রতিস্থাপন করা যাবে, তাতে সিস্টেমের আচরণ পরিবর্তন হবে না।**

যদি সাবক্লাস প্যারেন্টের মতো ব্যবহার করা না যায়, তাহলে ওটা LSP ভঙ্গ করছে।

🔹 **ভুল উদাহরণ (Bad Example):**

class Bird {

public function fly() {

return "Flying";

}

}

class Penguin extends Bird {

// পেঙ্গুইন উড়তে পারে না!

}

এখানে **Penguin** ক্লাসের জন্য **fly()** ফাংশন রাখা ঠিক না, কারণ পেঙ্গুইন উড়তে পারে না।

👉 **ভালো উদাহরণ (Good Example):**

interface Bird {

public function eat();

}

interface FlyingBird extends Bird {

public function fly();

}

class Sparrow implements FlyingBird {

public function eat() {

return "Eating";

}

public function fly() {

return "Flying";

}

}

class Penguin implements Bird {

public function eat() {

return "Eating";

}

}

এখানে **FlyingBird** ও **Bird** আলাদা করে পেঙ্গুইনের সমস্যা সমাধান করা হয়েছে।

**4. Interface Segregation Principle (ISP)**

**একটি ক্লায়েন্টকে এমন কোনো ইন্টারফেসে নির্ভরশীল করা যাবে না, যা সে ব্যবহার করে না।**

একটি বড় ইন্টারফেস থাকলে সেটাকে ছোট ছোট ইন্টারফেসে ভাগ করা উচিত।

🔹 **ভুল উদাহরণ (Bad Example):**

interface Worker {

public function work();

public function eat();

}

class HumanWorker implements Worker {

public function work() {

return "Working";

}

public function eat() {

return "Eating";

}

}

class RobotWorker implements Worker {

public function work() {

return "Working";

}

public function eat() {

// রোবট খায় না!

}

}

এখানে **RobotWorker** এর জন্য **eat()** অপ্রয়োজনীয়।

👉 **ভালো উদাহরণ (Good Example):**

interface Workable {

public function work();

}

interface Eatable {

public function eat();

}

class HumanWorker implements Workable, Eatable {

public function work() {

return "Working";

}

public function eat() {

return "Eating";

}

}

class RobotWorker implements Workable {

public function work() {

return "Working";

}

}

এখানে **Workable** ও **Eatable** আলাদা করে সমস্যা সমাধান করা হয়েছে।

**5. Dependency Inversion Principle (DIP)**

**হাই লেভেল মডিউলগুলো লো লেভেল মডিউলের উপর নির্ভরশীল হবে না। বরং উভয়েই অ্যাবস্ট্রাকশন (interface) এর উপর নির্ভর করবে।**

🔹 **ভুল উদাহরণ (Bad Example):**

class MySQLDatabase {

public function connect() {

return "Connected to MySQL";

}

}

class UserService {

private $db;

public function \_\_construct() {

$this->db = new MySQLDatabase(); // নির্দিষ্ট MySQL ডাটাবেজের উপর নির্ভরশীল

}

}

এখানে **UserService** সরাসরি **MySQLDatabase** এর উপর নির্ভর করছে, যা ডাটাবেজ পরিবর্তন করা কঠিন করে দেয়।

👉 **ভালো উদাহরণ (Good Example):**

interface Database {

public function connect();

}

class MySQLDatabase implements Database {

public function connect() {

return "Connected to MySQL";

}

}

class PostgreSQLDatabase implements Database {

public function connect() {

return "Connected to PostgreSQL";

}

}

class UserService {

private $db;

public function \_\_construct(Database $db) {

$this->db = $db; // এখন যে কোনো ডাটাবেজ ব্যবহার করা যাবে

}

}

এখন **UserService** যে কোনো ডাটাবেজ ব্যবহার করতে পারবে, শুধু **Database** ইন্টারফেস ইমপ্লিমেন্ট করলেই হবে।

**উপসংহার**

SOLID প্রিন্সিপল ব্যবহার করলে কোড আরও ক্লিন, মেইনটেনেবল ও ফ্লেক্সিবল হয়। এগুলো ফলো করলে সফটওয়্যার ডেভেলপমেন্ট সহজ হয়ে যায় এবং বাগ কমে যায়।

✅ **এই পাঁচটি নিয়ম মাথায় রেখে আপনি ভালো সফটওয়্যার ডিজাইন করতে পারবেন!** 🚀