Clean Code with Kotlin

SOLID Principles & Best Practices

Goal: to write maintainable, testable, and scalable software.

What is SOLID?

SOLID is an acronym for five design principles.

- **S** - Single Responsibility Principle
- **O** - Open/Closed Principle
- **L** - Liskov Substitution Principle
- **I** - Interface Segregation Principle
- **D** - Dependency Inversion Principle

Single Responsibility Principle (SRP)

> *A class should have one, and only one, reason to change.*

Instead of having a "God Class" that handles parsing, logic, and database operations, split these concerns into separate classes.

SRP in Kotlin: Bad vs. Good

Bad: Handles logic AND saving to a database.

```kotlin
class UserManager {
    fun createUser(name: String) {
        // Validation logic
        println("User $name created")
        // Database logic
        println("Saving $name to database...")
    }
}
```

Good: Responsibilities are separated.

```kotlin
class UserCreator {
    fun create(name: String): User = User(name)
}

class UserRepository {
    fun save(user: User) { /* Database logic */ }
}
```

Open/Closed Principle (OCP)

> *Software entities should be open for extension, but closed for modification.*

You should be able to add new functionality without altering existing code. In Kotlin, `interfaces` and `sealed classes` are excellent tools for this.

OCP in Kotlin: Bad vs. Good

Bad: Modifying the class for every new shape.

```kotlin
class AreaCalculator {
    fun calculate(shape: Any): Double = when (shape) {
        is Circle -> Math.PI * shape.radius * shape.radius
        is Square -> shape.side * shape.side
        else -> 0.0
    }
}
```

Good: Extending via an interface.

```kotlin
interface Shape { fun area(): Double }

class Circle(val radius: Double) : Shape {
    override fun area() = Math.PI * radius * radius
}

class AreaCalculator {
    fun calculate(shape: Shape): Double = shape.area()
}
```

The Expression Problem

Types vs. Operations

The Dilemma: How do you design code so you can add **new types** AND **new operations** without modifying existing code?

- **OOP** makes adding *Types* easy (new classes), but adding *Operations* hard (modifying all classes).
- **Functional/Kotlin** makes adding *Operations* easy (new functions), but *Types* hard (modifying all functions).

The Kotlin Solution: Use `sealed` interfaces to lock the types, making it easy to add new operations safely via exhaustive `when`.

```kotlin
sealed interface Expr
data class Value(val number: Int) : Expr
data class Sum(val left: Expr, val right: Expr) : Expr

fun eval(expr: Expr): Int = when (expr) {
    is Value -> expr.number
    is Sum -> eval(expr.left) + eval(expr.right)
}
fun printExpr(expr: Expr): String = when (expr) {
    is Value -> expr.number.toString()
    is Sum -> "${printExpr(expr.left)} + ${printExpr(expr.right)}"
}
```

Liskov Substitution Principle (LSP)

> *Derived classes must be substitutable for their base classes.*

If your code uses a base class, it should be able to use a subclass without knowing it, and without breaking the application.

LSP in Kotlin: Bad vs. Good

Bad: A Penguin is a Bird, but it can't fly. Breaking the contract!

```kotlin
open class Bird {
    open fun fly() { println("I am flying") }
}
class Penguin : Bird() {
    override fun fly() { throw Exception("I can't fly!") }
}
```

Good: Reorganize the hierarchy.

```
open class Bird
interface FlyingBird { fun fly() }

class Eagle : Bird(), FlyingBird {
    override fun fly() { println("Soaring!") }
}
class Penguin : Bird() // Doesn't implement FlyingBird
```

Interface Segregation Principle (ISP)

> *Make fine-grained interfaces that are client-specific.*

Clients shouldn't be forced to implement interfaces they don't use. Break fat interfaces into smaller, more specific ones.

ISP in Kotlin: Bad vs. Good

Bad: Forcing a robot to eat.

```kotlin
interface Worker {
    fun work()
    fun eat()
}
class Robot : Worker {
    override fun work() { println("Working...") }
    override fun eat() { /* I don't eat! */ }
}
```

Good: Smaller, focused interfaces.

```kotlin
interface Workable { fun work() }
interface Eatable { fun eat() }

class Robot : Workable {
    override fun work() { println("Working...") }
}
```

Dependency Inversion Principle (DIP)

> *Depend on abstractions, not on concretions.*

High-level modules should not depend on low-level modules. Both should depend on abstractions (interfaces).

DIP in Kotlin: Bad vs. Good

Bad: Hardcoded dependency on a specific database.

```
class MySQLDatabase {
    fun insert() { /* ... */ }
}
class UserService {
    val db = MySQLDatabase() // Tightly coupled!
}
```

Good: Depending on an interface.

```kotlin
interface Database { fun insert() }

class MySQLDatabase : Database {
    override fun insert() { /* ... */ }
}
class UserService(private val db: Database) {
    fun saveUser() { db.insert() }
}
```

General Best Practice: Constructor Injection

Never instantiate complex dependencies inside a class's `init` block or variable declarations.

Why?

- **Testing:** It's impossible to mock a dependency if the class creates it internally.
- **Flexibility:** You can't swap out implementations (e.g., swapping a `NetworkDataSource` for a `MockDataSource`).

Constructor Injection: Bad vs. Good

Bad: The dependency is hardcoded inside the class.

```
class PaymentProcessor {
    // Hard to test, impossible to change without modifying this class
    private val apiClient = StripeApiClient()

    fun process() { apiClient.charge() }
}
```

Good: The dependency is passed in (Injected).

```
class PaymentProcessor(private val apiClient: ApiClient) {
    fun process() { apiClient.charge() }
}

// In your tests, you can now do:
// val processor = PaymentProcessor(MockApiClient())
```

Summary

- **S**: One reason to change.
- **O**: Extend, don't modify.
- **L**: Subclasses should behave like their parents.
- **I**: Keep interfaces small and specific.
- **D**: Depend on abstractions (interfaces).
- **Constructor Injection:** Pass dependencies in, don't build them inside.