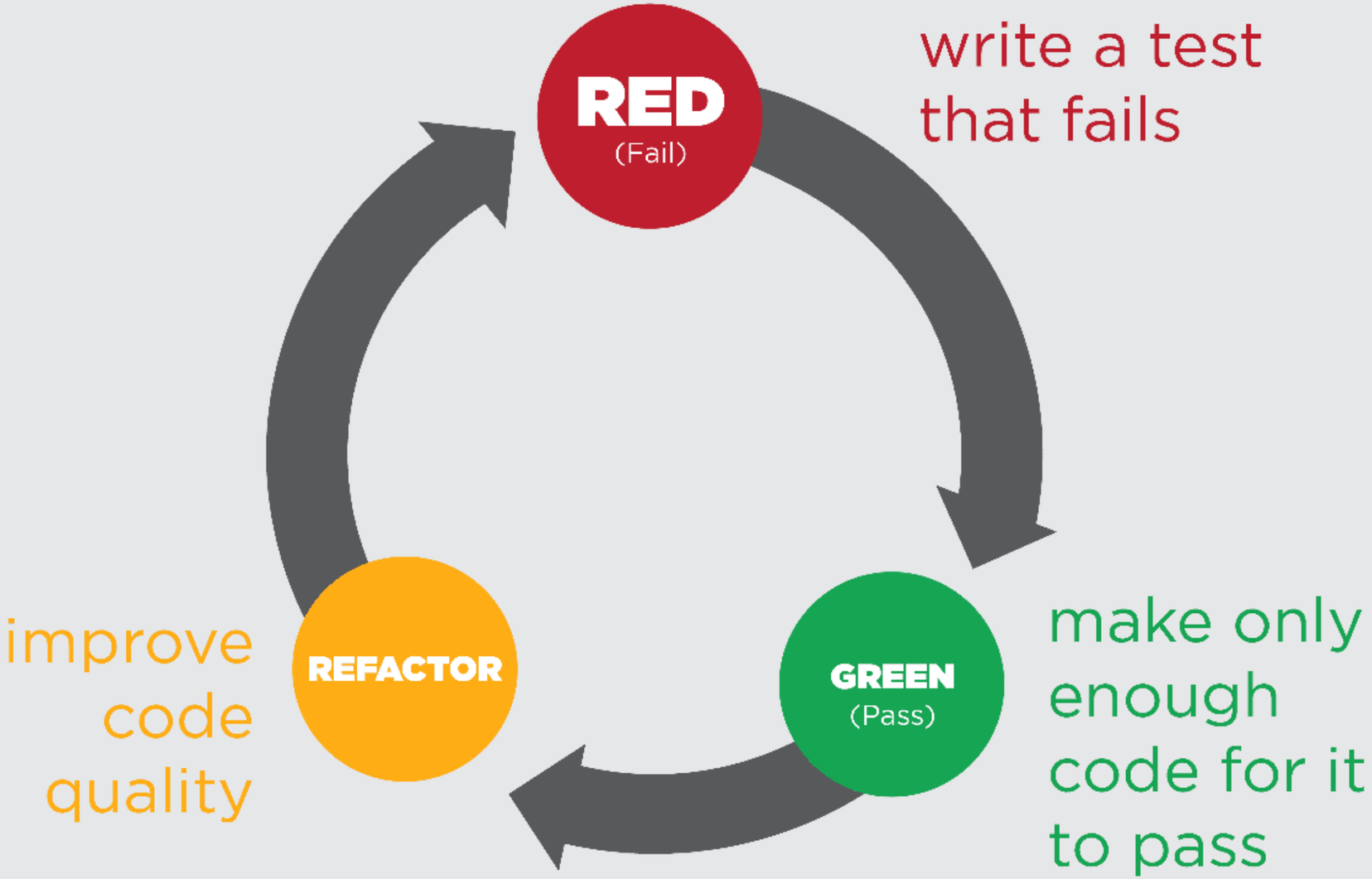
Test-Driven Development (TDD)



```
fun testAddition() {
    val calculator = Calculator()
    assertEquals(5.0, calculator.add(2.0, 3.0))
    assertEquals(-3.0, calculator.add(-1.0, -2.0))
    assertEquals(10.0, calculator.add(0.0, 10.0))
}
```

```
fun testSubtraction() {
    val calculator = Calculator()
    assertEquals(2.0, calculator.subtract(5.0, 3.0))
    assertEquals(1.0, calculator.subtract(-1.0, -2.0))
    assertEquals(-2.0, calculator.subtract(3.0, 5.0))
}
```

```
fun testMultiplication() {
    val calculator = Calculator()
    assertEquals(6.0, calculator.multiply(2.0, 3.0))
    assertEquals(0.0, calculator.multiply(10.0, 0.0))
    assertEquals(6.0, calculator.multiply(-2.0, -3.0))
}
```

```
fun testDivision() {
    val calculator = Calculator()
    assertEquals(2.0, calculator.divide(6.0, 3.0))
    assertEquals(0.5, calculator.divide(1.0, 2.0))
    try {
        calculator.divide(10.0, 0.0)
        fail("Should have thrown an ArithmeticException")
    } catch (e: ArithmeticException) {
        // Expected exception
    }
}
```

```
class Calculator {

    fun add(a: Int, b: Int): Int = a + b

    fun subtract(a: Int, b: Int): Int = a - b

    fun multiply(a: Int, b: Int): Int = a * b

    fun divide(a: Int, b: Int): Int = if (b != 0) a / b else throw IllegalArgumentException("Division by zero")
}
```

```
fun testDiscountedRate() {
    val calculator = Calculator()
    // simple discounted price based on an original price and a discount percentage
    assertEquals(80, calculator.calculateDiscountedRate(100, 20))   // 20% discount from 100
    assertEquals(45, calculator.calculateDiscountedRate(50, 10))    // 10% discount from 50
    assertEquals(180, calculator.calculateDiscountedRate(200, 10))  // 10% discount from 200
    assertEquals(396, calculator.calculateDiscountedRate(440, 10))  // 10% discount from 440
    assertEquals(950, calculator.calculateDiscountedRate(1000, 5))  // 5% discount from 1000
}
```

```
fun testNetPresentValue() {
    val calculator = Calculator()
    // Net Present Value (NPV) of a future cash flow over 5 periods
    assertEquals(952, calculator.calculateNetPresentValue(1000, 0.05, 1))  // Future value of 1000 after 1 year at 5%
    assertEquals(907, calculator.calculateNetPresentValue(950, 0.06, 2))   // Future value of 950 after 2 years at 6%
    assertEquals(863, calculator.calculateNetPresentValue(900, 0.04, 3))   // Future value of 900 after 3 years at 4%
    assertEquals(822, calculator.calculateNetPresentValue(850, 0.07, 4))   // Future value of 850 after 4 years at 7%
    assertEquals(783, calculator.calculateNetPresentValue(800, 0.05, 5))   // Future value of 800 after 5 years at 5%
}
```

```
class Calculator {

    // Previously implemented "basic" operations...

    fun calculateDiscountedRate(originalPrice: Int, discountPercent: Int): Int {
        return originalPrice - (originalPrice * discountPercent / 100)
    }

    fun calculateNetPresentValue(futureValue: Int, rate: Double, periods: Int): Int {
        val discountFactor = Math.pow(1 + rate, periods.toDouble())
        val presentValue = futureValue / discountFactor
        return presentValue.toInt()
    }
}
```

```
class CalculatorTest {

    private val originalOut = System.out
    private val outputStreamCaptor = ByteArrayOutputStream()

    @BeforeEach
    fun setUp() {
        System.setOut(PrintStream(outputStreamCaptor))
    }

    @AfterEach
    fun tearDown() {
        System.setOut(originalOut)
    }

    @Test
    fun testAddition() {
        val calculator = Calculator(ConsoleUI())  // Assuming ConsoleUI for output
        calculator.add(2.0, 3.0)
        assertEquals("Result: 5.0", outputStreamCaptor.toString().trim())
        outputStreamCaptor.reset() // Clear output for next test
        calculator.add(-1.0, -2.0)
        assertEquals("Result: -3.0", outputStreamCaptor.toString().trim())
    }

    @Test
    fun testSubtraction() {
        val calculator = Calculator(ConsoleUI())
        calculator.subtract(5.0, 3.0)
        assertEquals("Result: 2.0", outputStreamCaptor.toString().trim())
    }

    ...
```

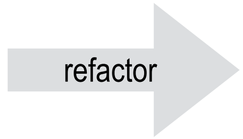
```
class Calculator {

    // Previously implemented "basic" operations...

    fun calculateDiscountedRate(originalPrice: Int, discountPercent: Int): Int {
        return originalPrice - (originalPrice * discountPercent / 100)
    }

    fun calculateNetPresentValue(futureValue: Int, rate: Double, periods: Int): Int {
        val discountFactor = Math.pow(1 + rate, periods.toDouble())
        val presentValue = futureValue / discountFactor
        return presentValue.toInt()
    }
}

    fun calculateNetPresentValue(futureValue: Int, rate: Double, periods: Int): Int {
        val discountFactor = Math.pow(1 + rate, periods.toDouble())
        val presentValue = futureValue / discountFactor
        return presentValue.toInt()
    }
}
```

```
class Calculator {

    fun add(a: Int, b: Int): Int = a + b

    fun subtract(a: Int, b: Int): Int = a - b

    fun multiply(a: Int, b: Int): Int = a * b

    fun divide(a: Int, b: Int): Int = if (b != 0) a / b
          else throw IllegalArgumentException("Division by zero")

    fun calculateDiscountedRate(originalPrice: Int, discountPercent: Int): Int {
        return originalPrice - (originalPrice * discountPercent / 100)
    }

    fun calculateNetPresentValue(futureValue: Int, rate: Double, periods: Int): Int {
        val discountFactor = Math.pow(1 + rate, periods.toDouble())
        val presentValue = futureValue / discountFactor
        return presentValue.toInt()
    }
}
```

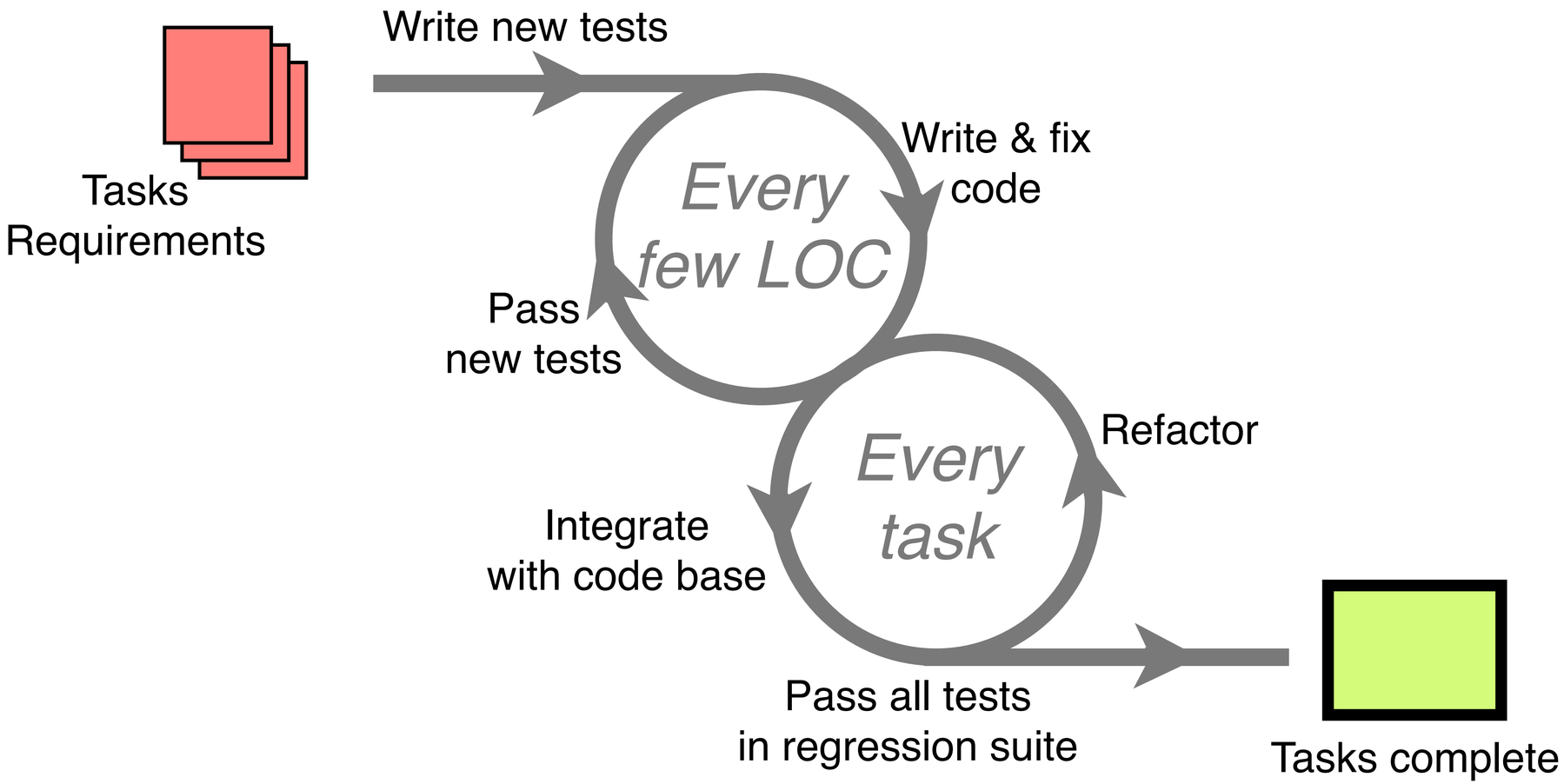


```
interface CalculatorUI {
    fun display(result: String)
}

class Calculator(private val ui: CalculatorUI) {
    fun add(a: Int, b: Int) {
        ui.display("Sum: ${a + b}")
    }

    fun subtract(a: Int, b: Int) {
        ui.display("Difference: ${a - b}")
    }

    fun multiply(a: Int, b: Int) {
        ui.display("Product: ${a * b}")
    }

    fun divide(a: Int, b: Int) {
        if (b == 0) {
            ui.display("Error: Division by zero is undefined.")
        } else {
            ui.display("Quotient: ${a / b}")
        }
    }

    // the other operations go here ...
}
```

```
interface CalculatorUI {
    fun display(result: String)
}

class Calculator(private val ui: CalculatorUI) {
    fun add(a: Int, b: Int) {
        ui.display("Sum: ${a + b}")
    }

    fun subtract(a: Int, b: Int) {
        ui.display("Difference: ${a - b}")
    }

    fun multiply(a: Int, b: Int) {
        ui.display("Product: ${a * b}")
    }

    fun divide(a: Int, b: Int) {
        if (b == 0) {
            ui.display("Error: Division by zero is undefined.")
        } else {
            ui.display("Quotient: ${a / b}")
        }
    }

    // the other operations go here ...
}
```




```
abstract class Operation {
    abstract fun perform(a: Int, b: Int): Int
}

class AddOperation : Operation() {
    override fun perform(a: Int, b: Int): Int = a + b
}

class SubtractOperation : Operation() {
    override fun perform(a: Int, b: Int): Int = a - b
}

class MultiplyOperation : Operation() {
    override fun perform(a: Int, b: Int): Int = a * b
}

// the other operations go here ...

interface CalculatorUI {
    fun display(result: String)
}

class Calculator(private val ui: CalculatorUI) {
    fun executeOperation(a: Int, b: Int, operation: Operation) {
        val result = operation.perform(a, b)
        ui.display("Result: $result")
    }
}
```



Adapted from Nagappan et al., "Realizing quality improvement through TDD: results and experiences of four industrial teams"

TDD Trade-Offs

- Benefits
  - *Cleaner code, fewer bugs, line coverage > 90%*
  - *Every line of code you write serves a clear purpose*
  - *Typically the resulting code is more modular and flexible without overengineering*
  - *Tests serve as documentation for expected behavior*
  - *Comprehensive tests gives you confidence to refactor code*
- Drawbacks
  - *Can slow down development, especially at the beginning*
  - *Can make code too reactive to (overfit) the tests*
  - *There is quite a learning curve for doing TDD effectively*
  - *Writing tests before code requires an upfront time investment*

WHEN To Use TDD?

- Good candidates
  - *user interface behavior (button enabling, button logic, models, etc.)*
  - *business logic*
  - *pretty much any Java/Kotlin class / method*
- Bad candidates
  - *user interface appearance (layout, colors, etc.)*
  - *client/server interactions (will need to do mock testing)*
  - *large code bases, legacy code*

TDD

RED (Fail)

write a test that fails

GREEN (Pass)

make only enough code for it to pass

REFACTOR

improve code quality

https://www.rdegra.com/tdd-is-not-about-testing-but-the-design/