EPFL

CS-311: Testing Fundamentals

Prof. George Candea
School of Computer & Communication Sciences

Recap of Testing

Testing: Different Levels

(recap from CS-214)

- Unit tests
  - *individual components or functions (e.g., a user input validation function of the app)*
- Integration tests
  - *test combinations of components (e.g., login interface + backend database)*
- System tests
  - *the complete and integrated software (e.g., testing the entire app's functionality and its interaction with Android and the sensors)*
- Acceptance tests
  - *testing the product in real-world scenarios to ensure it meets user requirements (e.g., alpha and beta testing with a group of end-users)*

```
data class CartItem(val name: String, val price: Double, val quantity: Int)

class Order(val items: List<CartItem>) {
    fun calculateTotal(): Double = items.sumOf { it.price * it.quantity }

    fun addItem(item: CartItem) {
        ...
    }

    fun removeItem(itemName: String) {
        ...
    }
}
```


```
@Test
fun calculateTotalReturnsCorrectTotal() {
    val items = listOf(
        CartItem("Pizza", 10.0, 2),
        CartItem("Burger", 5.0, 3)
    )
    val total = calculateTotal(items)
    assertEquals(35.0, total, 0.001)
}
```

Unit test

Writing Good Unit Tests

- Small and check only one thing
  - *each unit test focused on at most one functionality*
- Independent of each other
- Decouple (as much as possible) from the rest of the implementation
  - *especially when it comes to object creation*


```
data class CartItem(val name: String, val price: Double, val quantity: Int)

class Order(val items: List<CartItem>) {
    fun calculateTotal(): Double = items.sumOf { it.price * it.quantity }

    fun addItem(item: CartItem) {
        ...
    }

    fun removeItem(itemName: String) {
        ...
    }
}

interface OrderRepository {
    fun saveOrder(order: Order): Order // Return the saved order
    fun findById(orderId: Int): Order?
    fun updateOrderStatus(order: Order, newStatus: OrderStatus): Boolean
    fun findOrdersByUser(userId: Int): List<Order>
}

interface PaymentProcessor {
    fun processPayment(order: Order, paymentInfo: PaymentInfo): PaymentResult
}

data class PaymentInfo(val cardNumber: String, val expiry: String, val cvv: Int, /* ... */)
data class PaymentResult(val status: PaymentStatus, val transactionId: String? = null)
```

```
@Test
fun successfulOrderPlacement() {
    val pizza = CartItem("Pizza", 10.0, 2)
    val burger = CartItem("Burger", 5.0, 3)
    val order = Order(listOf(pizza, burger))

    val orderRepository = OrderRepositoryImpl() // Connected to a test database
    val paymentProcessor = TestPaymentProcessor() // Connected to a payment sandbox

    val orderService = OrderService(orderRepository, paymentProcessor)
    val orderResult = orderService.placeOrder(order)

    assertEquals(OrderStatus.CONFIRMED, orderResult.status)
    val savedOrder = orderRepository.findById(orderResult.orderId)
    assertNotNull(savedOrder)

    orderRepository.deleteOrder(orderResult.orderId) // Cleanup: remove the test order
}
```

Integration test


```
class MainScreen : Screen<MainScreen>() {
    val menuButton = KButton { withId(R.id.open_menu_button) }
    val pizzaItem = KTextView { withText("Pizza") }
    // ...
}

@Test
fun successfulOrderAndDeliveryUpdateTest() {
    before {
        // ...
    }.after {
        // ...
    }

    Scenario("Order Placement with Search and History Check") {
        MainScreen {
            searchButton.click()
            searchEditText.typeText("Luigi's Pizza")
            searchResult("Luigi's Pizza").click()

            // ... (Add items to cart, checkout)

            // Navigate to order history
            orderHistoryButton.click()
            orderHistoryList {
                firstChild<KTextView> { withText("Pizza, Burger - CHF 35.00") }
                isVisible() // Check that the element is visible on the screen
            }
        }
    }
}
```

System test


```
Feature: Successful Order Placement

  As a hungry user, I want to easily order food from my favorite restaurants
  so that I can satisfy my cravings without hassle.

  Scenario: Placing a Basic Order
    Given I am on the PolyFood app's main screen
    And I have selected a restaurant
    When I add a "Pizza" (price CHF 10.00) with quantity 2 to my cart
    And I add a "Burger" (price CHF 5.00) with quantity 3 to my cart
    And I proceed to checkout
    And I enter valid payment details and a delivery address
    And I confirm the order
    Then I should see an order confirmation message
    And the order should appear in my order history with the status "Preparing"
```

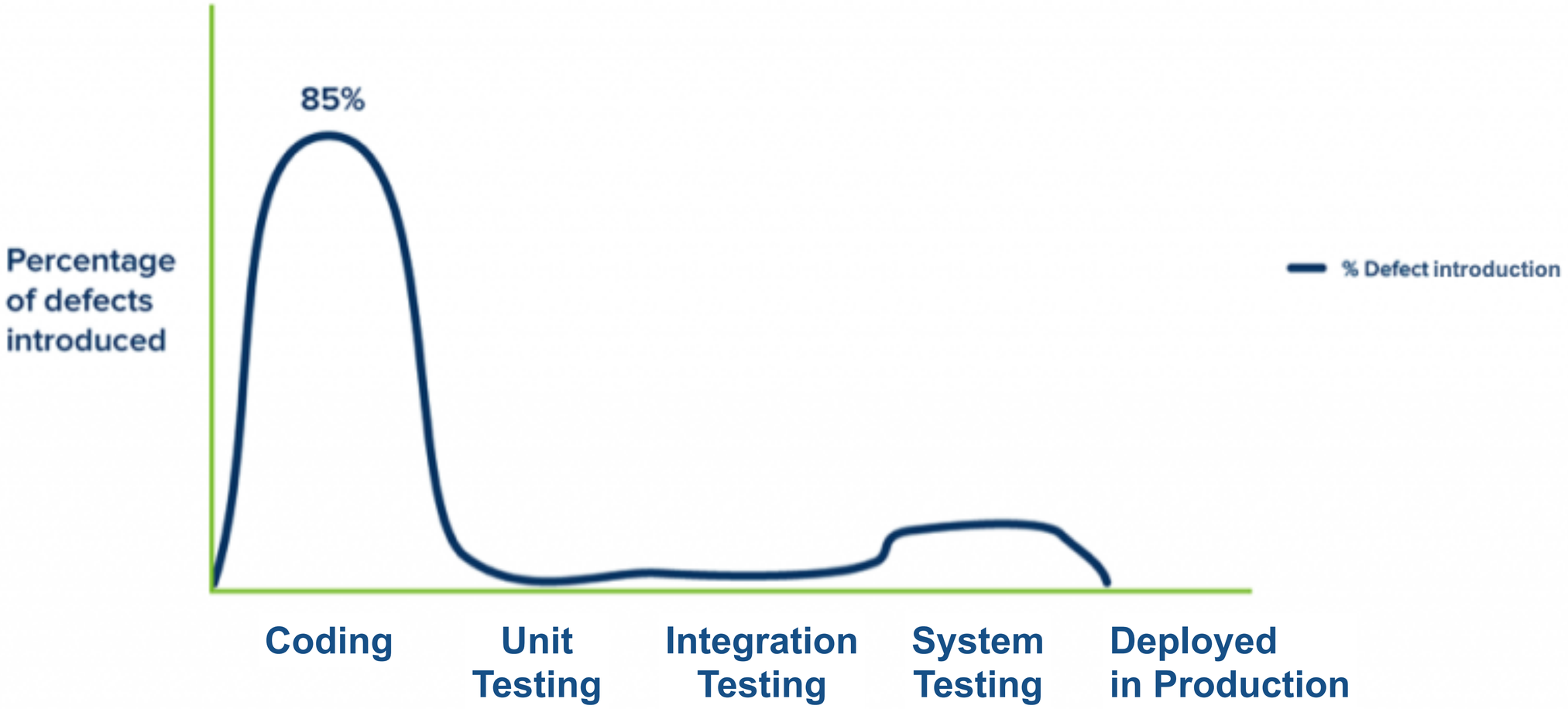
Acceptance test

Testing: Different Levels

(recap from CS-214)

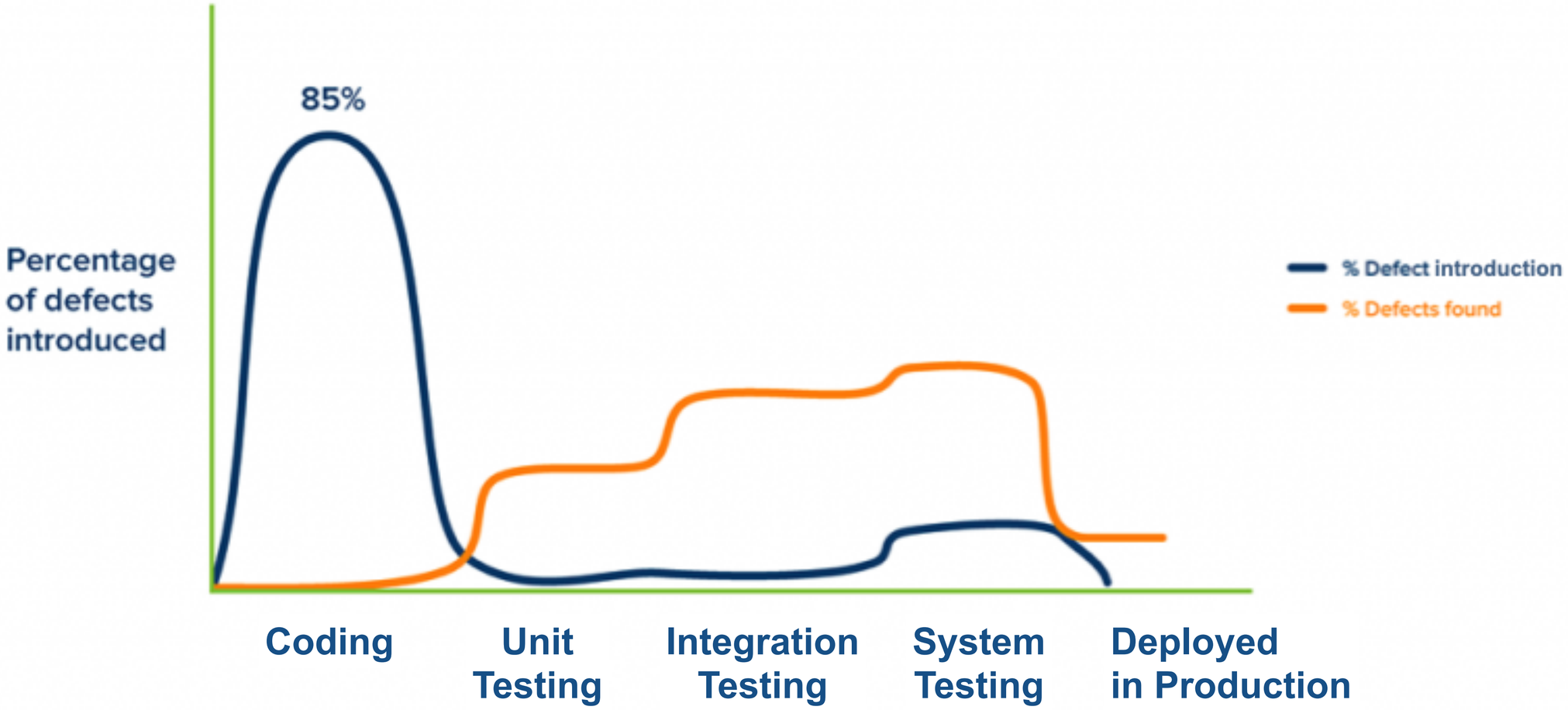
- Unit tests
  - *individual components or functions (e.g., a user input validation function of the app)*
- Integration tests
  - *test combinations of components (e.g., login interface + backend database)*
- System tests
  - *the complete and integrated software (e.g., testing the entire app's functionality and its interaction with Android and the sensors)*
- Acceptance tests
  - *testing the product in real-world scenarios to ensure it meets user requirements (e.g., alpha and beta testing with a group of end-users)*

Cost of Bugs



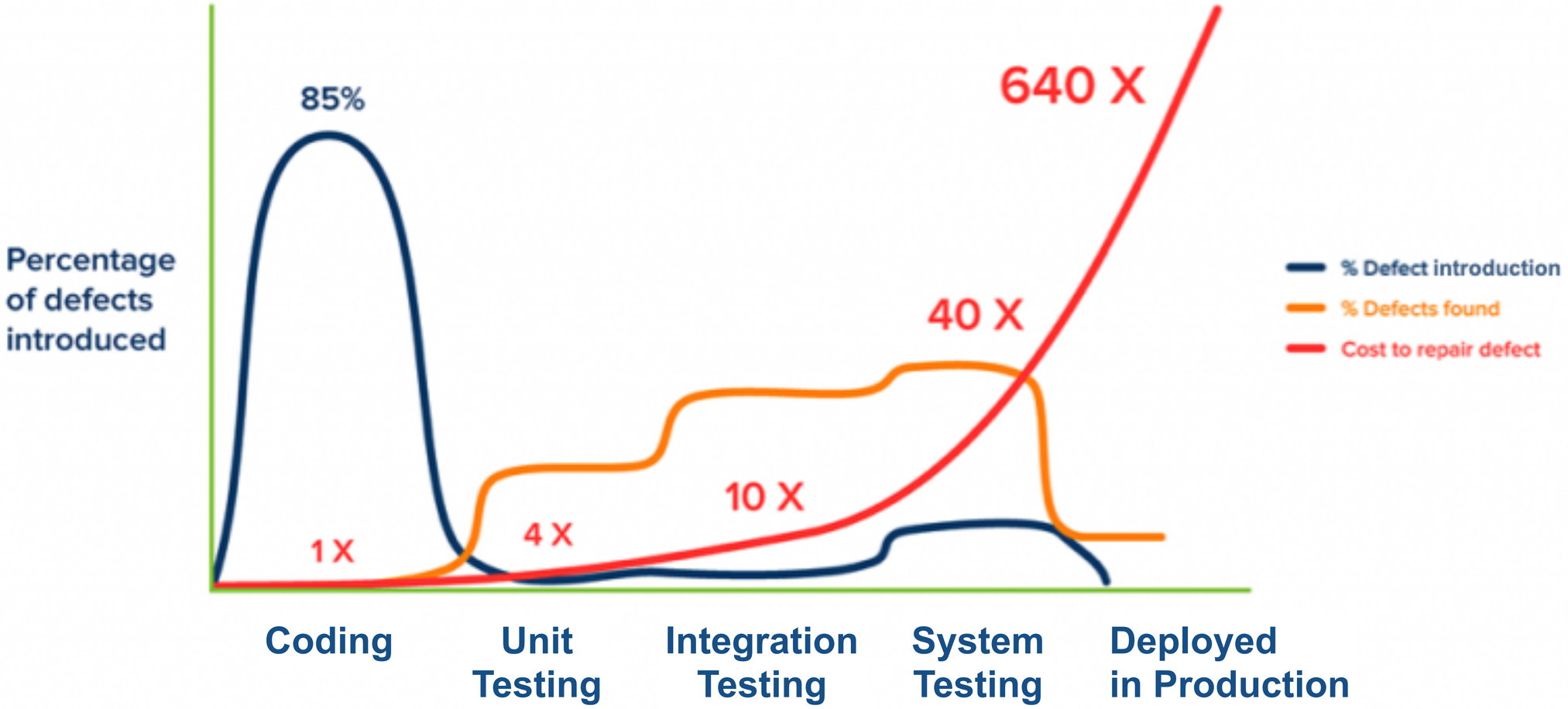
Jones, Capers. *Applied Software Measurement: Global Analysis of Productivity and Quality.*

https://www.stickyminds.com/article/shift-left-approach-software-testing



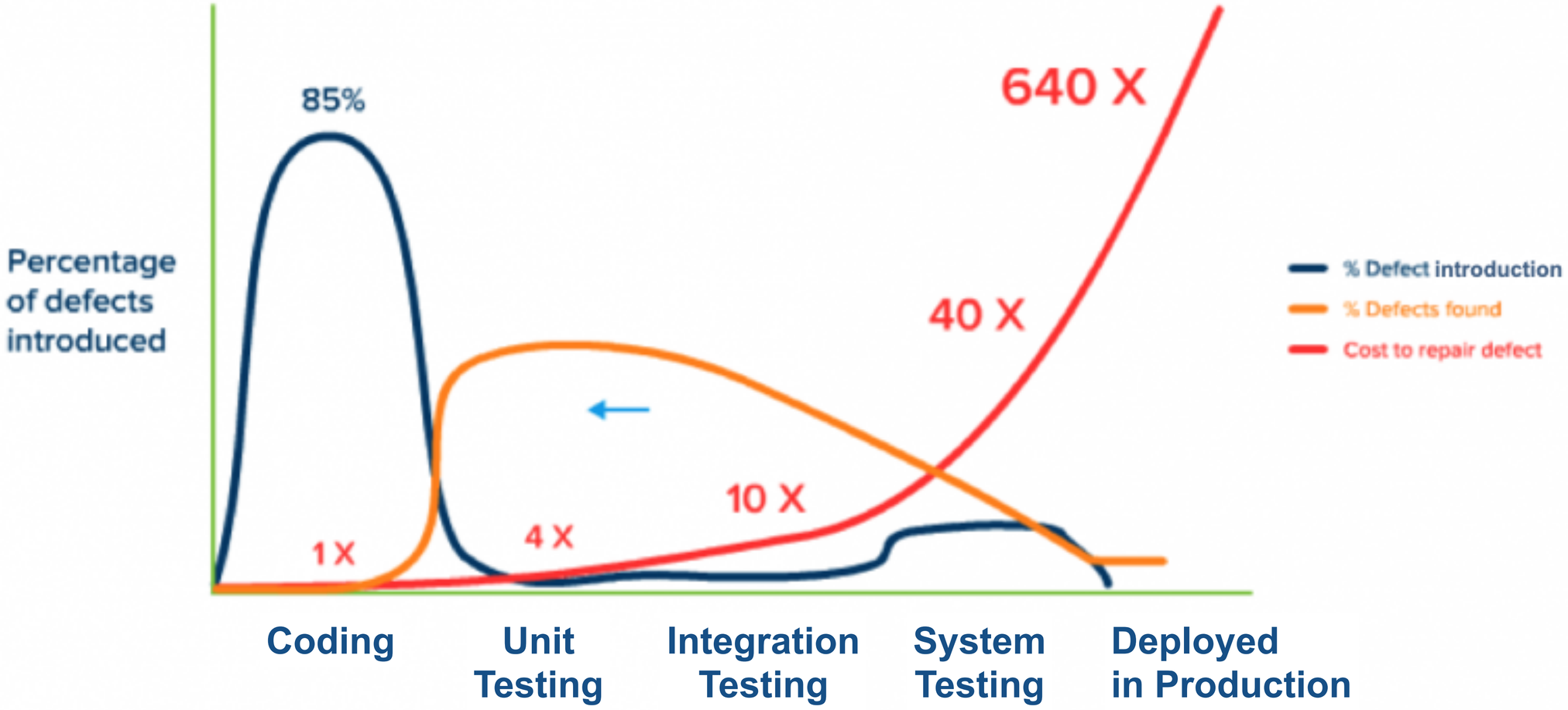
Jones, Capers. *Applied Software Measurement: Global Analysis of Productivity and Quality.*

https://www.stickyminds.com/article/shift-left-approach-software-testing

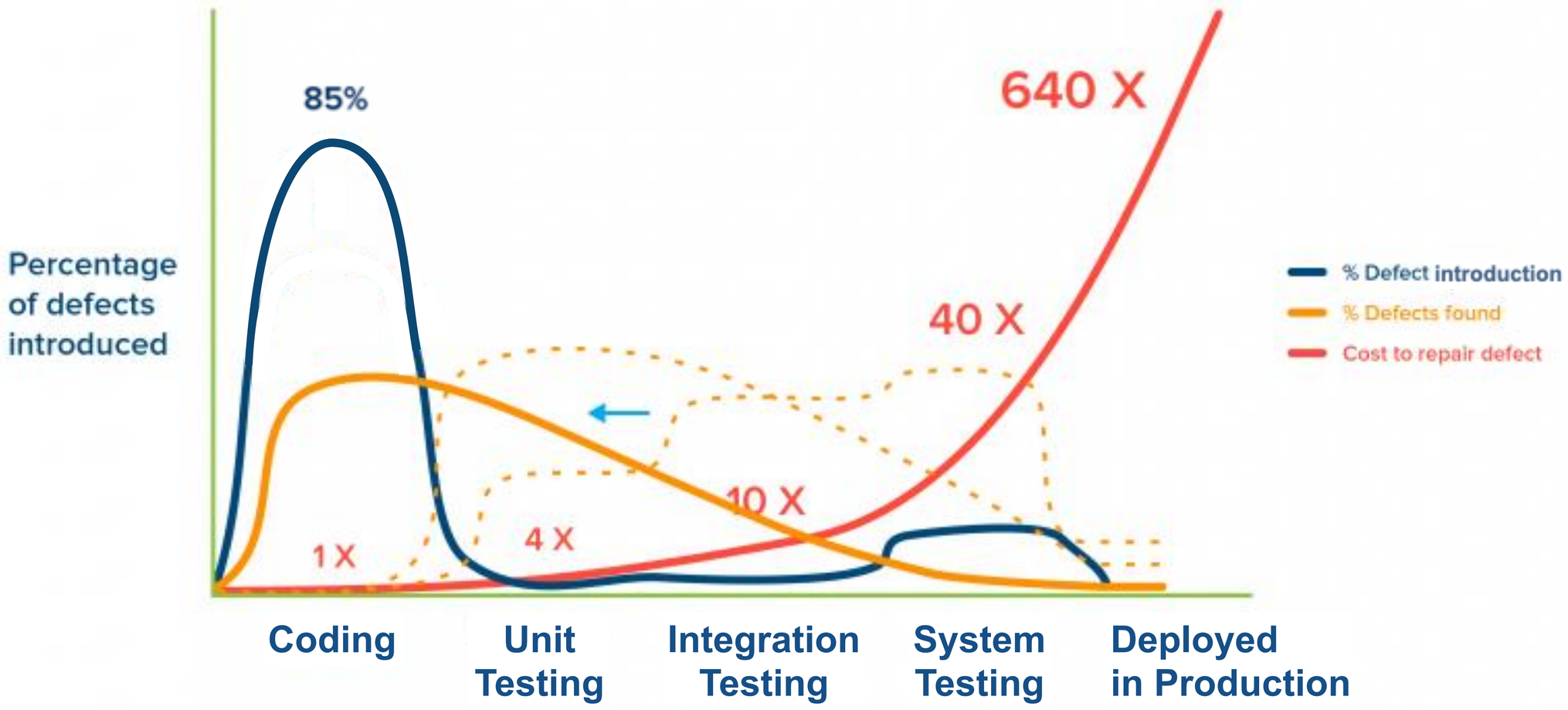


Jones, Capers. *Applied Software Measurement: Global Analysis of Productivity and Quality.*

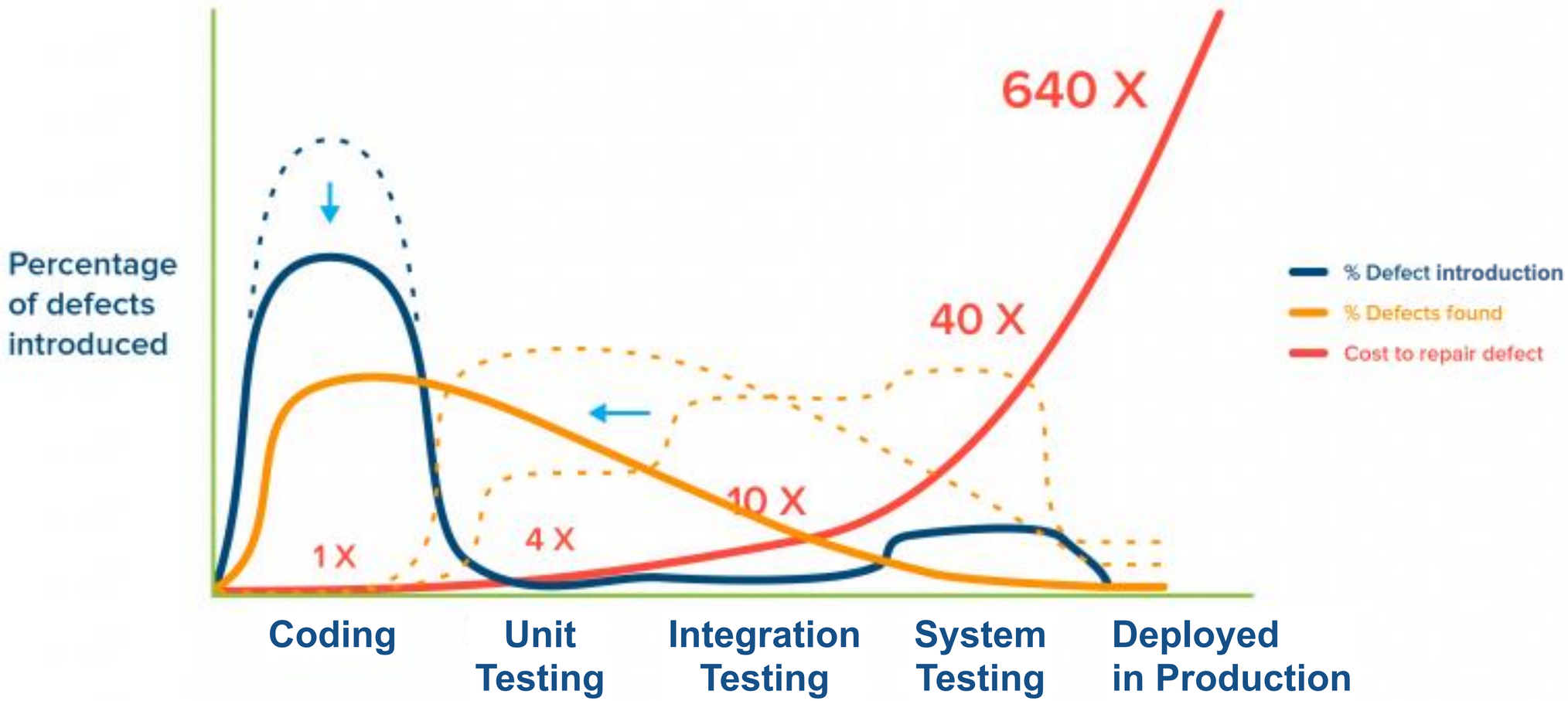
https://www.stickyminds.com/article/shift-left-approach-software-testing



Jones, Capers. *Applied Software Measurement: Global Analysis of Productivity and Quality.*



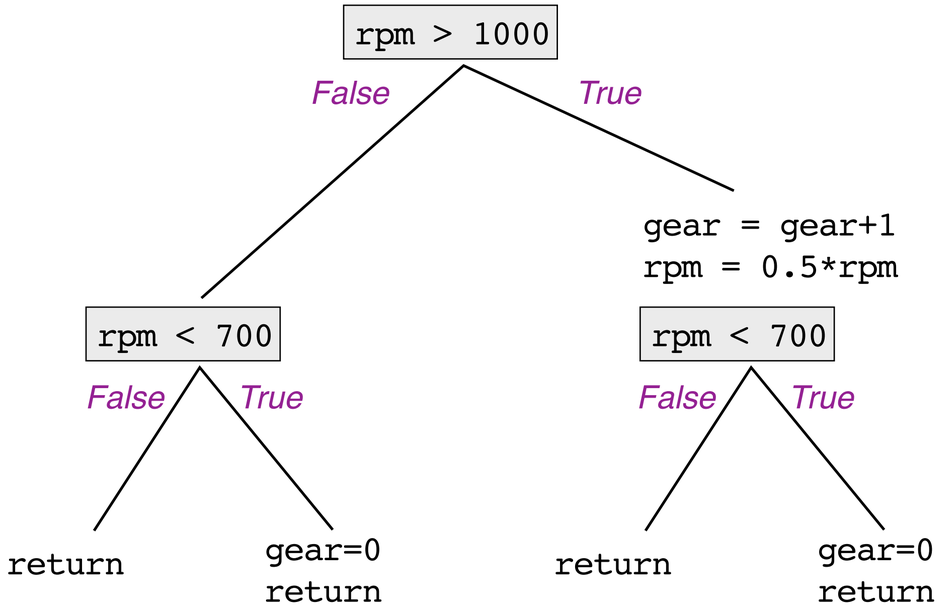
Jones, Capers. *Applied Software Measurement: Global Analysis of Productivity and Quality.*



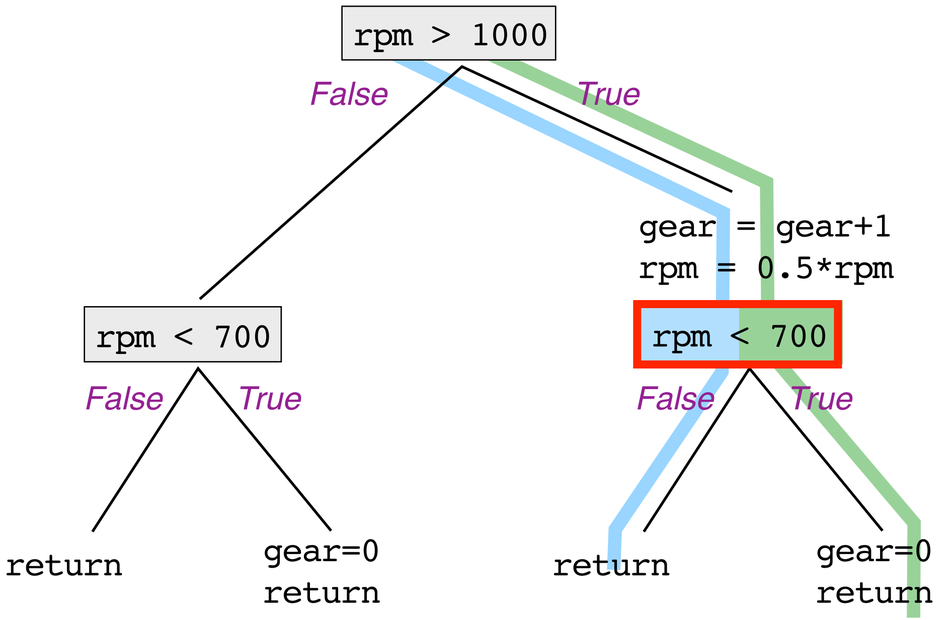
Jones, Capers. *Applied Software Measurement: Global Analysis of Productivity and Quality.*

How well can we test?

```
autoShift (int rpm)
   if (rpm > 1000)
      gear = gear+1
      rpm = 0.5*rpm
   if (rpm < 700)
      gear=0
   return
```



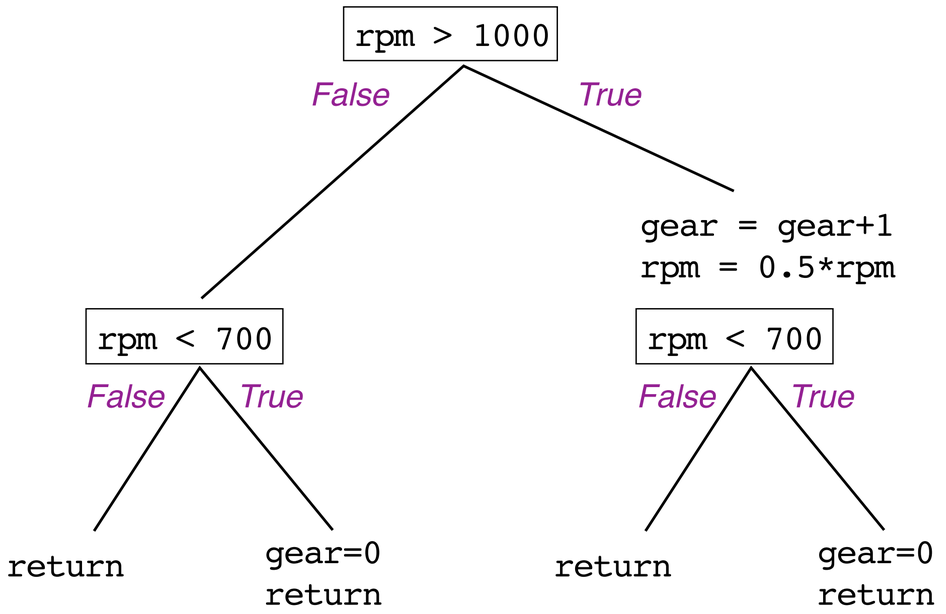
```
autoShift (int rpm)
    if (rpm > 1000)
        gear = gear+1
        rpm = 0.5*rpm
    if (rpm < 700)
        gear=0
    return
```



Software Testing

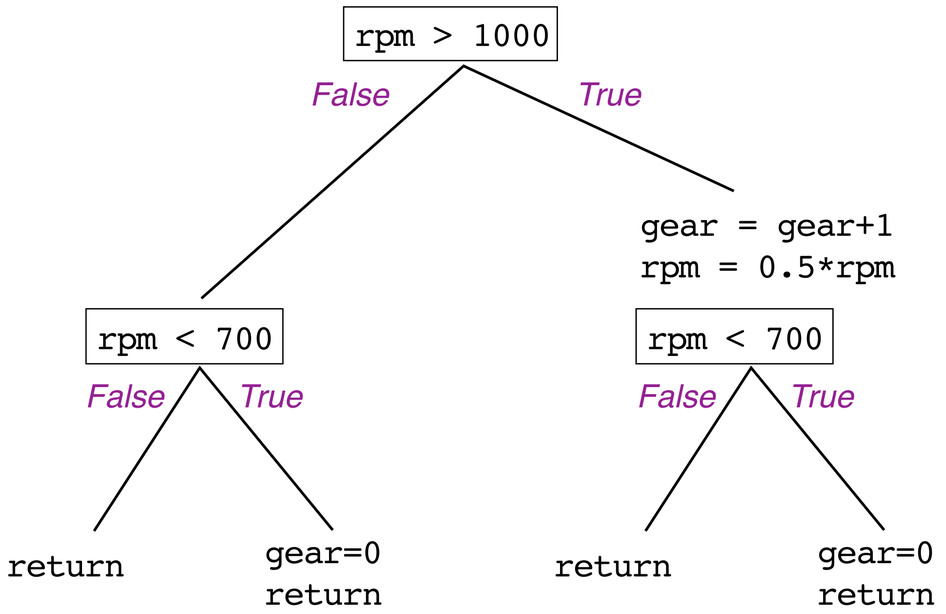
rpm ∈ {0,350,944,1200,1800}

```
autoShift (int rpm)
    if (rpm > 1000)
        gear = gear+1
        rpm = 0.5*rpm
    if (rpm < 700)
        gear=0
    return
```



Software Testing

```
autoShift (int rpm)
    if (rpm > 1000)
        gear = gear+1
        rpm = 0.5*rpm
    if (rpm < 700)
        gear=0
    return
```



$$paths \simeq 2^{program\ size}$$

How many possible execution paths (behaviors) ?

2

```
char evenOdd (int num)
{
    if (num % 2 == 0) {
        return 'E'; // even
    } else {
        return 'O'; // odd
    }
}
```

3

```
double max (double n1, double n2, double n3)
{
    if (n1 >= n2 & n1 >= n3)
        return n1;
    else if (n2 >= n1 & n2 >= n3)
        return n2;
    else
        return n3;
}
```

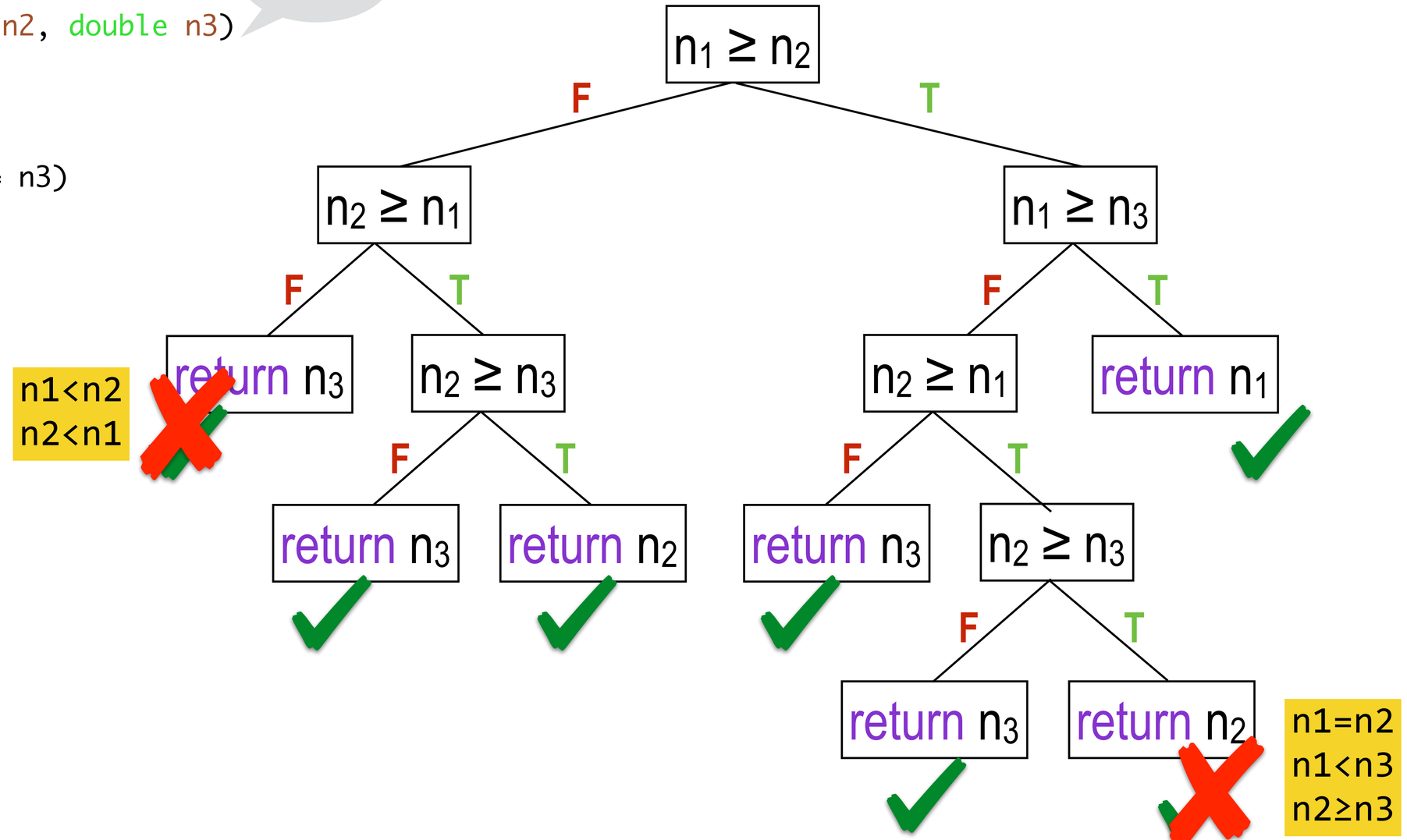
```
double max (double n1, double n2, double n3)
{
    if (n1 >= n2 && n1 >= n3)
        return n1;
    else if (n2 >= n1 && n2 >= n3)
        return n2;
    else
        return n3;
}
```

short-circuit evaluation

How many possible execution paths (behaviors) ?

```
double max (double n1, double n2, double n3)
{
    if (n1 >= n2 && n1 >= n3)
        return n1;
    else if (n2 >= n1 && n2 >= n3)
        return n2;
    else
        return n3;
}
```

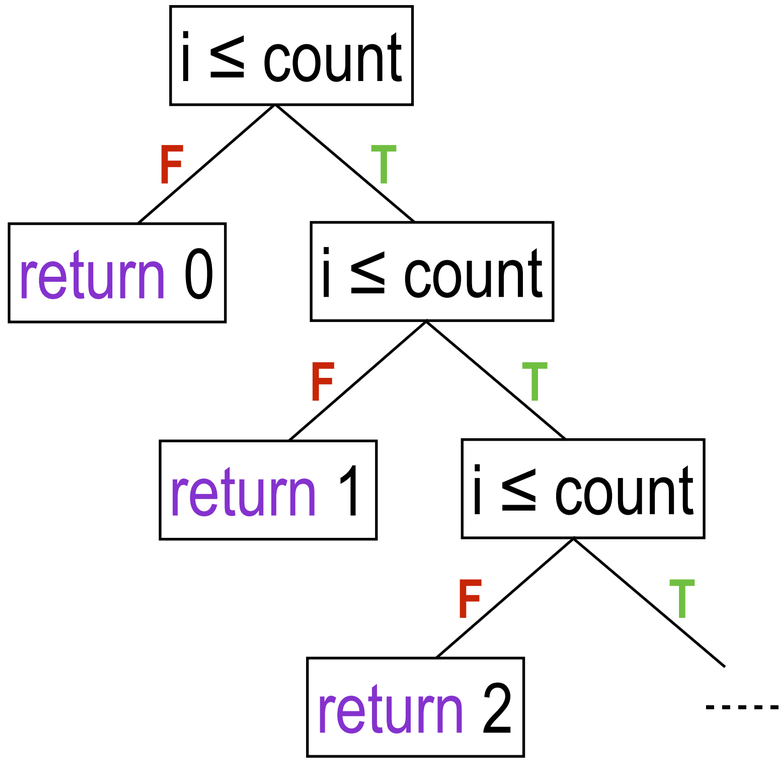
5



How many execution paths (possible behaviors) ?

128

```
int iterationsByte (byte count)
{
    int iterations=0;
    for (int i = 1; i <= count; ++i) {
        ++iterations;
    }
    return iterations;
}
```



How many execution paths (possible behaviors) ?

```
int iterationsShort (short count)
{
    int iterations=0;
    for (int i = 1; i <= count; ++i) {
        ++iterations;
    }
    return iterations;
}
```

32768

```
void foo (String language1, String language2)
{
    LinkedList<String> languages = new LinkedList<>();

    languages.add(language1);
    languages.add(language2);
    System.out.println("LinkedList: " + languages);
}
```

?

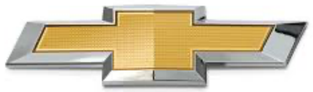
have no idea how many possible paths in here

Some Code Sizes (in LOC)

- smartphone app w/ social networking etc. ~10s-100s of KLOC

- Boeing 787 avionics + online support ~several million LOC (MLOC)

- Chrome browser ~several MLOC



- entry-level electric vehicle (Chevy Volt) ~10 MLOC

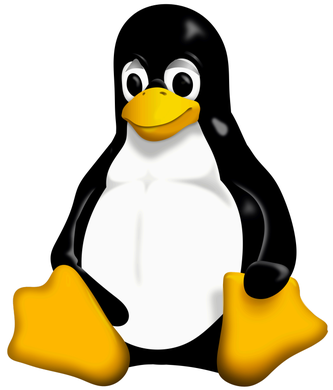
- Android operating system ~a few tens of MLOC

- the Large Hadron Collider ~50 MLOC

- software of a self-driving car ~100 MLOC

- all Google services combined ~2 billion LOC (2'000 MLOC)

Software Testing



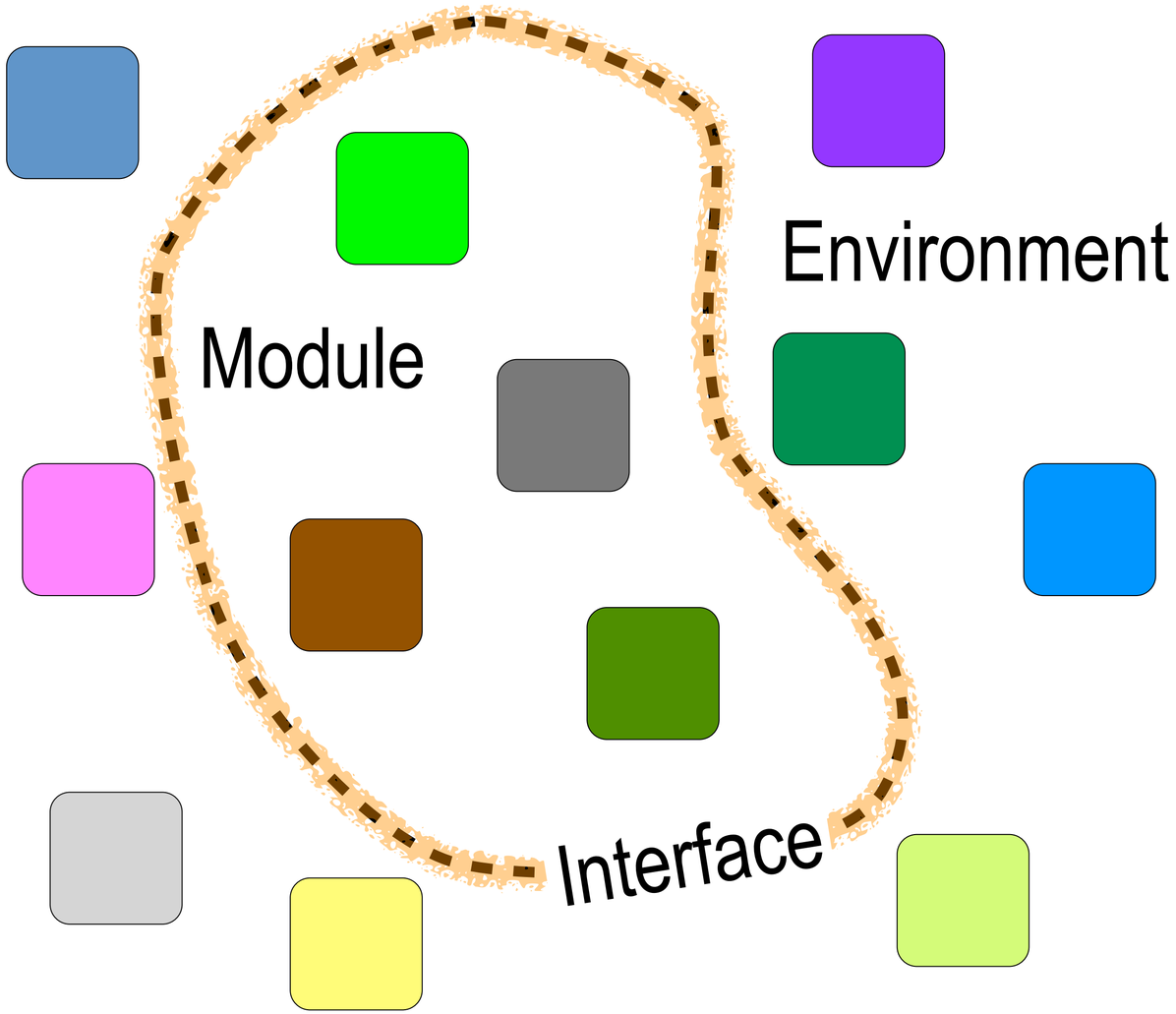
way more than 5,000,000 lines of code[1] (LOC) ⇒ > $2^{500,000}$ paths

Can we test $2^{500,000}$ paths?

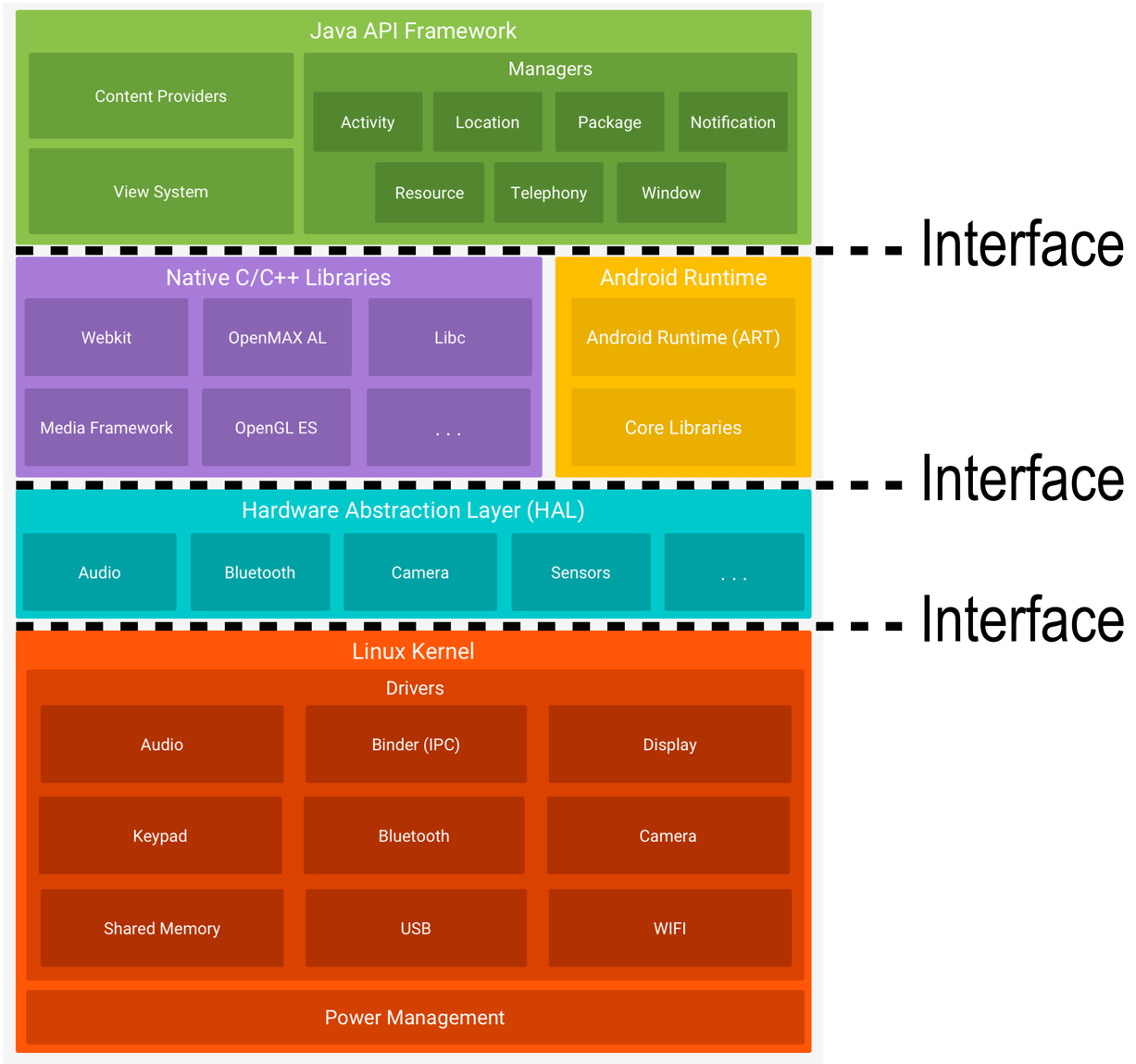
[1] https://www.visualcapitalist.com/millions-lines-of-code/

Testing Programs ≠ Testing Software

Modularity



Layering



https://developer.android.com/guide/platform

Conclusion

- Cannot test all behaviors (too many)
- Goal of testing
  - *identify bugs in order to fix them*
  - *gain confidence in software quality*
  - *statistically verify that program meets requirements*
- How do we tell how much testing we still need to do?

> Program testing can be used to show the presence of bugs, but never to show their absence!
>
> *Edsger W. Dijkstra*
> *"Notes on Structured Programming" (1970)*

Test-Quality Metrics

```kotlin
@Test
fun testApplyThreshold() {
    val items = listOf(Item("Soup", 5.0))
    val threshold = 10.0
    val weeklyDiscounts = emptyMap<String, Double>() // no discounts
    val result = totalAfterDiscounts(items, threshold, weeklyDiscounts)

    assertEquals(threshold, result)  // should return min cost not soup cost
}
```

```kotlin
@Test
fun testSingleItemWithDiscount() {
    val items = listOf(Item("Steak", 40.0))
    val weeklyDiscounts = mapOf("Steak" to 0.8)
    val result = totalAfterDiscounts(items, 30.0, weeklyDiscounts)

    assertEquals(32.0, result)  // should return 40 * 0.8 = 32
}
```

```kotlin
fun totalAfterDiscounts(items: List<Item>, threshold: Double,
                        weeklyDiscounts: Map<String, Double>): Double {
    var discountedTotal = 0.0
    for (item in items) {
        val discountFactor = weeklyDiscounts.getOrDefault(item.name, 1.0)
        val itemPrice = item.price * discountFactor
        discountedTotal += itemPrice
    }

    if (discountedTotal > minCost) {
        return discountedTotal
    } else {
        return threshold
    }
}
```

statement coverage = 8 / 8 = 100%

```kotlin
@Test
fun testApplyThreshold() {
    val items = listOf(Item("Soup", 5.0))
    val threshold = 10.0
    val weeklyDiscounts = emptyMap<String, Double>() // no discounts
    val result = totalAfterDiscounts(items, threshold, weeklyDiscounts)

    assertEquals(threshold, result)  // should return min cost not soup cost
}
```

```kotlin
@Test
fun testSingleItemWithDiscount() {
    val items = listOf(Item("Steak", 40.0))
    val weeklyDiscounts = mapOf("Steak" to 0.8)
    val result = totalAfterDiscounts(items, 30.0, weeklyDiscounts)

    assertEquals(32.0, result)  // should return 40 * 0.8 = 32
}
```

```kotlin
@Test
fun testNoItems() {
    val items = listOf<Item>()
    val threshold = 50.0
    val weeklyDiscounts = emptyMap<String, Double>()
    val result = totalAfterDiscounts(items, threshold, weeklyDiscounts)

    assertEquals(threshold, result)
}
```

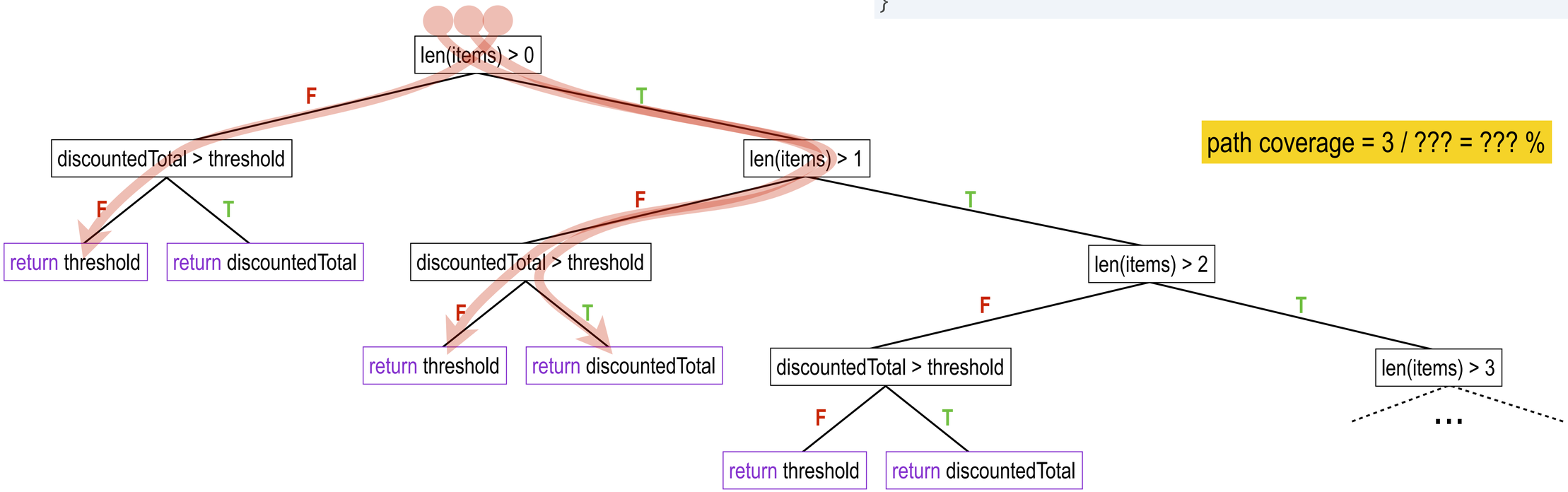
```kotlin
fun totalAfterDiscounts(items: List<Item>, threshold: Double,
                        weeklyDiscounts: Map<String, Double>): Double {
    var discountedTotal = 0.0
    for (item in items) {
        val discountFactor = weeklyDiscounts.getOrDefault(item.name, 1.0)
        val itemPrice = item.price * discountFactor
        discountedTotal += itemPrice
    }

    if (discountedTotal > threshold) {
        return discountedTotal
    } else {
        return threshold
    }
}
```

branch coverage = 4 / 4 = 100%

⚠ *Map<K,V>.getOrDefault()* implies a branch: *item.name* is found or not found. Here, we abstract that concern away.

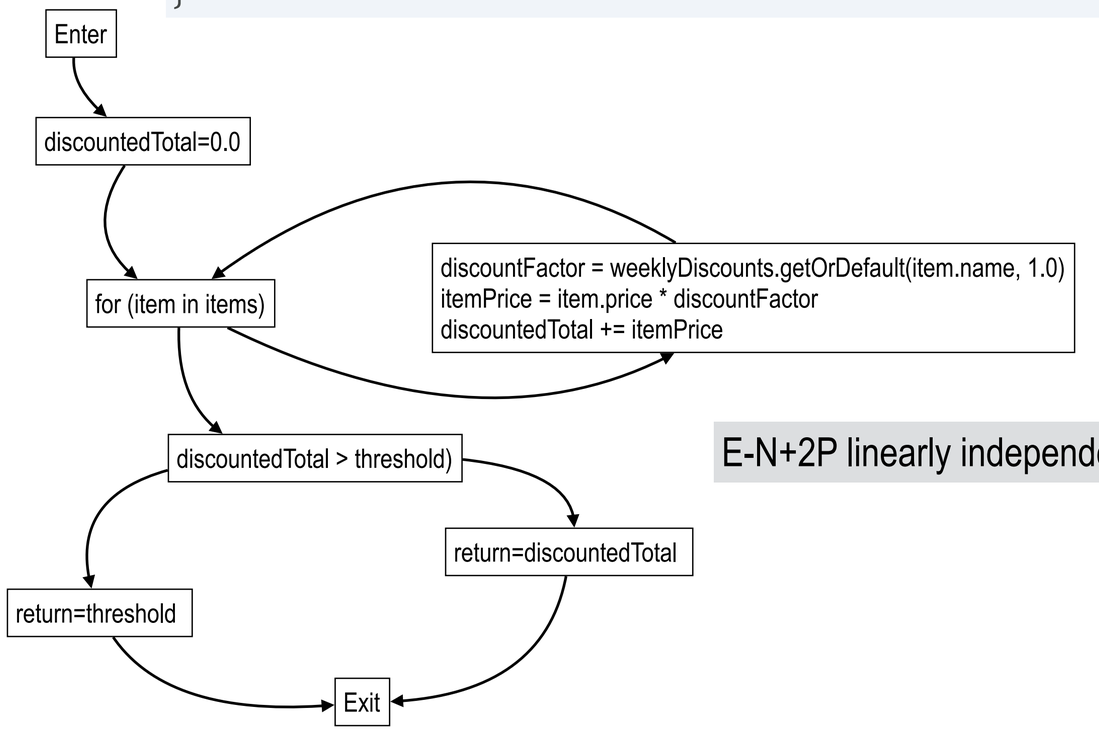
```
@Test
fun testApplyThreshold() {
    val items = listOf(Item("Soup", 5.0))
@Test
fun testSingleItemWithDiscount() {
@Test
fun testNoItems() {
    val items = listOf<Item>()
    val threshold = 50.0
    val weeklyDiscounts = emptyMap<String, Double>()
    val result = totalAfterDiscounts(items, threshold, weeklyDiscounts)

    assertEquals(threshold, result)
}
```

```
fun totalAfterDiscounts(items: List<Item>, threshold: Double,
                        weeklyDiscounts: Map<String, Double>): Double {
    var discountedTotal = 0.0
    for (item in items) {
        val discountFactor = weeklyDiscounts.getOrDefault(item.name, 1.0)
        val itemPrice = item.price * discountFactor
        discountedTotal += itemPrice
    }

    if (discountedTotal > threshold) {
        return discountedTotal
    } else {
        return threshold
    }
}
```



path coverage = 3 / ??? = ??? %

```
fun totalAfterDiscounts(items: List<Item>, threshold: Double,
                        weeklyDiscounts: Map<String, Double>): Double {
    var discountedTotal = 0.0
    for (item in items) {
        val discountFactor = weeklyDiscounts.getOrDefault(item.name, 1.0)
        val itemPrice = item.price * discountFactor
        discountedTotal += itemPrice
    }

    if (discountedTotal > threshold) {
        return discountedTotal
    } else {
        return threshold
    }
}
```



E-N+2P linearly independent paths

```kotlin
@Test
fun testApplyThreshold() {
    val items = listOf(Item("Soup", 5.0))
    val threshold = 10.0
    val weeklyDiscounts = emptyMap<String, Double>() // no discounts
    val result = totalAfterDiscounts(items, threshold, weeklyDiscounts)

    assertEquals(threshold, result)  // should return min cost not soup cost
}
```

```kotlin
@Test
fun testSingleItemWithDiscount() {
    val items = listOf(Item("Steak", 40.0))
    val weeklyDiscounts = mapOf("Steak" to 0.8)
    val result = totalAfterDiscounts(items, 30.0, weeklyDiscounts)

    assertEquals(32.0, result)  // should return 40 * 0.8 = 32
}
```

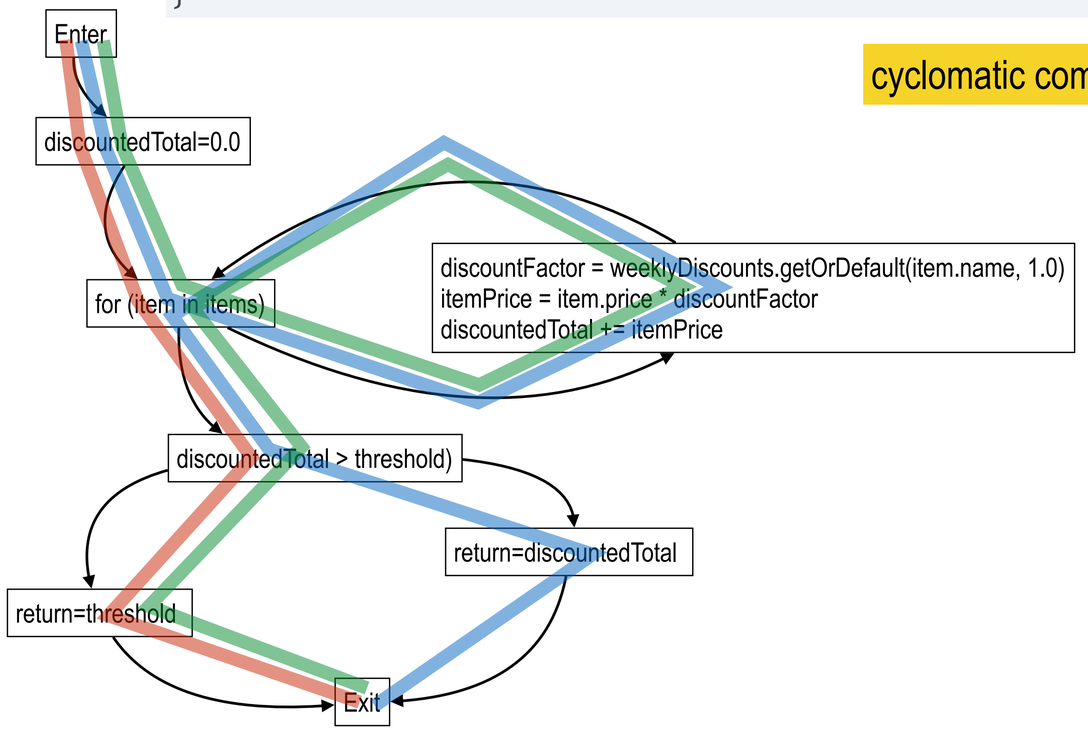
```kotlin
@Test
fun testNoItems() {
    val items = listOf<Item>()
    val threshold = 50.0
    val weeklyDiscounts = emptyMap<String, Double>()
    val result = totalAfterDiscounts(items, threshold, weeklyDiscounts)

    assertEquals(threshold, result)
}
```

1

+1

+1

```kotlin
fun totalAfterDiscounts(items: List<Item>, threshold: Double,
                        weeklyDiscounts: Map<String, Double>): Double {
    var discountedTotal = 0.0
    for (item in items) {
        val discountFactor = weeklyDiscounts.getOrDefault(item.name, 1.0)
        val itemPrice = item.price * discountFactor
        discountedTotal += itemPrice
    }

    if (discountedTotal > threshold) {
        return discountedTotal
    } else {
        return threshold
    }
}
```



cyclomatic complexity = 3

Recap: Test-Quality Metrics

- Statement coverage
  - *easy to compute, easy to reason about*
  - *a decent first-cut approximation of how good your tests are*
- Branch coverage
  - *more complicated but still doable*
  - *a better estimate of the quality of your tests*
  - *basis-set testing (cyclomatic complexity) gives us the sets for 100% branch coverage*
- Path coverage
  - *impractical*
  - *the perfect measure of how good your tests are*

Outline

- Recap of Testing
- Cost of Bugs
  - *the later you eliminate the defect, the more expensive it is*
- How well can we test?
  - *"Testing can be used to show the presence of bugs, never their absence!"*
  - *Measure coverage: statement / branch / path*
- Coverage Metrics in Practice
- Test-Driven Development (TDD)
- Behavior-Driven Development (BDD)

Coverage Metrics in Practice

Representative Example: JaCoCo

- = automated code coverage tool for Java/Kotlin
- Uses bytecode instrumentation
  - *injects probes in your compiled code*
  - *counters track specific events*
- Collects the data at run time
- Report generation
  - *user-friendly reports (HTML, XML, etc.) to visualize lines and branches covered*
- Integration
  - *Maven, Gradle, SonarCloud, ...*

JaCoCo Instrumentation (conceptual)

```
class JacocoDemo {
    fun isEven(number: Int): Boolean {
        return if (number % 2 == 0) {
            println("Number is even")
            true
        } else {
            println("Number is odd")
            false
        }
    }

    fun printSomething() {
        println("foo")
    }
}

fun main() {
    val jacocoDemo = JacocoDemo()

    println(jacocoDemo.isEven(89))
    jacocoDemo.printSomething()
}
```

```
class JacocoDemo {
    fun isEven(number: Int): Boolean {
➡       JaCoCo.recordExecution(1) // Record method entry (probe #1)

        return if (number % 2 == 0) {
➡           JaCoCo.recordExecution(2) // Record execution (probe #2)
            println("Number is even")
            true
        } else {
➡           JaCoCo.recordExecution(3) // Record execution (probe #3)
            println("Number is odd")
            false
        }
    }

    fun printSomething() {
➡       JaCoCo.recordExecution(4) // Record method entry (probe #4)
        println("foo")
    }
}

fun main() {
➡JaCoCo.recordExecution(5) // Record method entry (probe #5)
    val jacocoDemo = JacocoDemo()

    println(jacocoDemo.isEven(89))
    jacocoDemo.printSomething()
➡JaCoCo.recordExecution(6) // Record method exit (probe #6)
}
```

JacocoDemo

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
default		26%		100%	2	5	6	8	2	4	1	2
Total	31 of 42	26%	0 of 2	100%	2	5	6	8	2	4	1	2

default

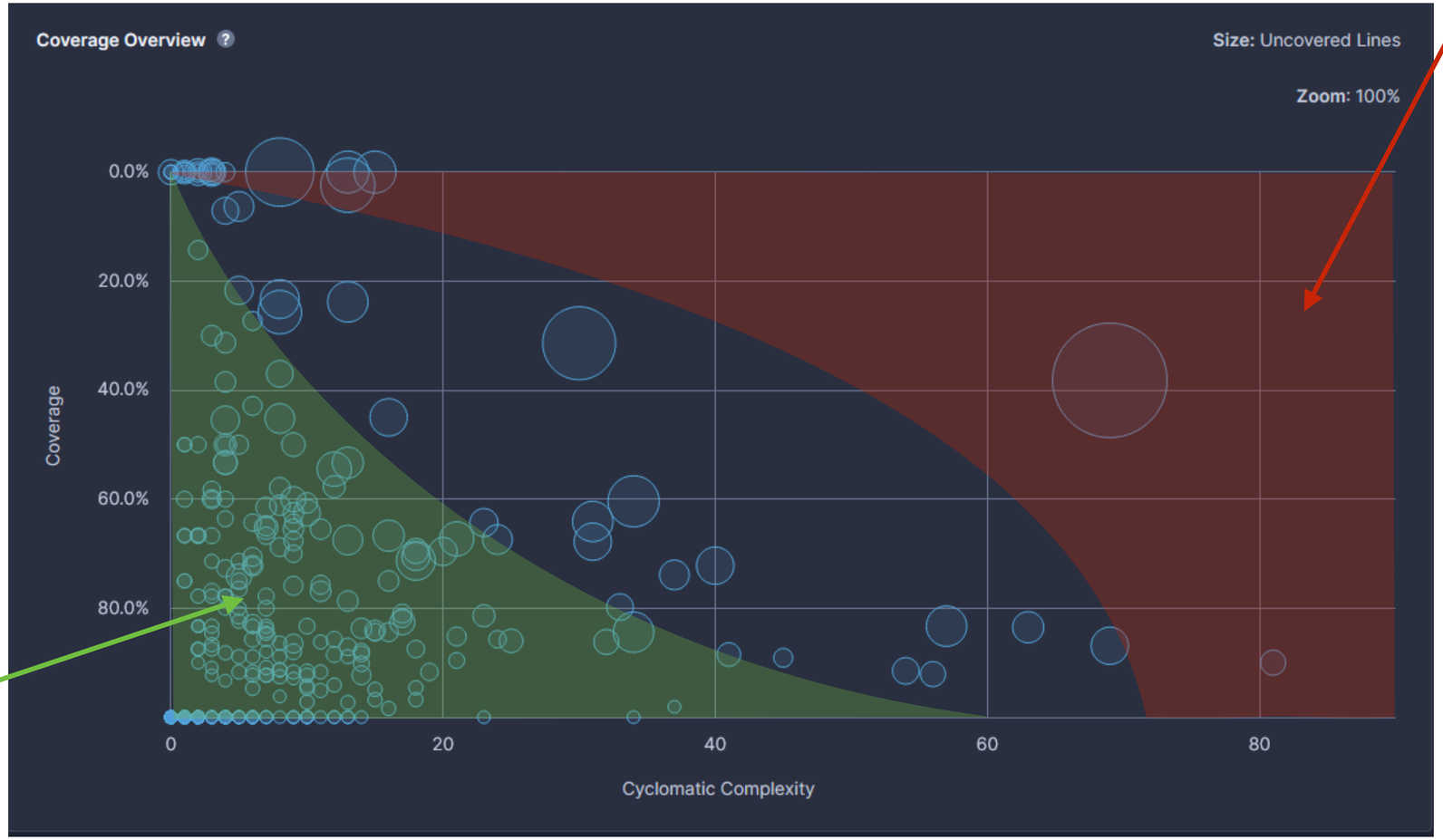
Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
MainKt		0%		n/a	1	1	4	4	1	1	1	1
JacocoDemo		57%		100%	1	4	2	4	1	3	0	1
Total	31 of 42	26%	0 of 2	100%	2	5	6	8	2	4	1	2

Main.kt

```
1.  class JacocoDemo {
2.      fun isEven(number: Int): Boolean {
3.          return if(number % 2 == 0){
4.              println("Number is even")
5.              true
6.          } else {
7.              println("Number is odd")
8.              false
9.          }
10.     }
11.
12.     fun printSomething() {
13.         println("foo")
14.     }
15. }
16.
17. fun main() {
18.     val jacocoDemo = JacocoDemo()
19.
20.     println(jacocoDemo.isEven(89))
21.     println(jacocoDemo.printSomething())
22. }
23.
```

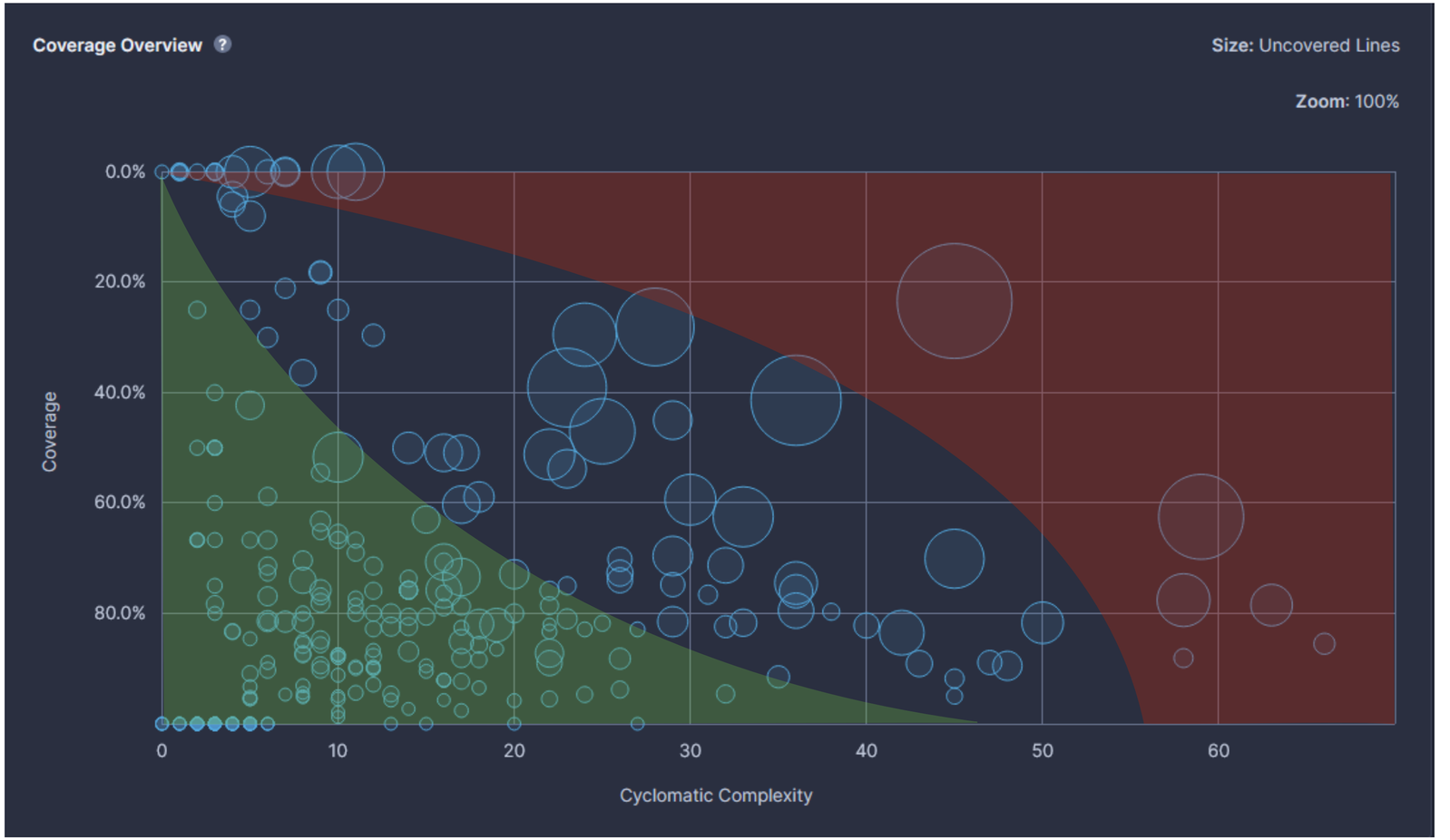
Tool Example: Sonar Cloud

Bad:
High Complexity/
Low Coverage

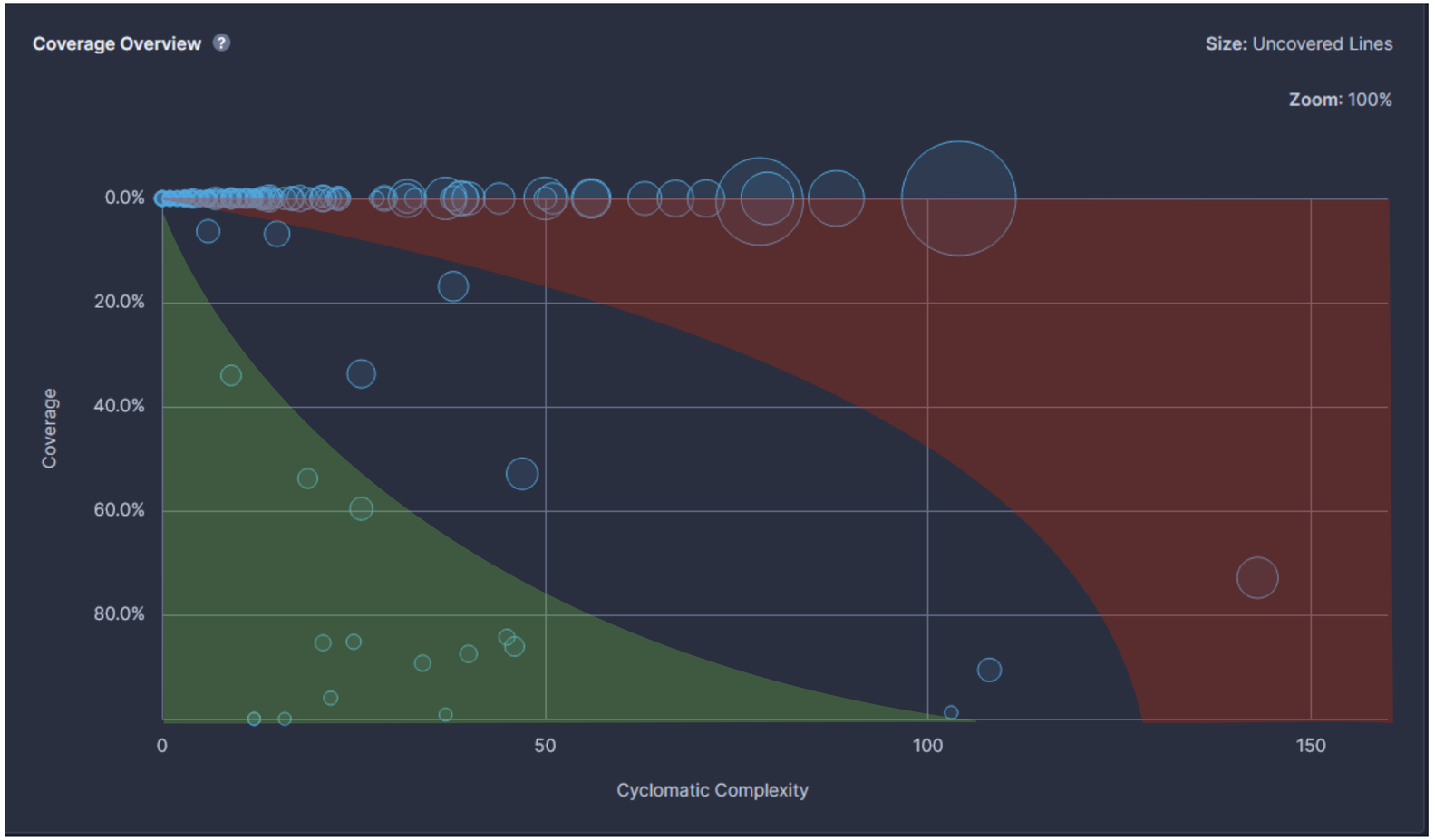


Good:
Low Complexity/
High Coverage

Coverage vs. Complexity



Coverage vs. Complexity



Coverage vs. Technical Debt

Technical Debt

- Implied cost of additional rework caused by doing a hack
  - *fixing hard-coded values, poorly structured code, coding-convention violations, ...*
- Causes
  - *Rushed timelines, incomplete requirements, lack of knowledge, legacy code, evolution*
- Kinds of technical debt
  - *Deliberate, Inadvertent, Bit Rot*
- Managing technical debt
  - *Recognize, Measure, Devise a repayment plan, Prevent*

Coverage vs. Technical Debt

LOC vs. Technical Debt

Code Smells

- Code patterns that suggest a potential issue or problem
  - *not bugs, just indicators of increased risks for future bugs, maintainability challenges, etc.*

Large Class
Long Method
Primitive Obsession
Feature Envy
Duplicated Code
God Object
Data Clumps

Shotgun Surgery
Lazy Class/Freeloader
Speculative Generality
Temporary Field
Message Chains
Middle Man

LOC vs. Technical Debt

LOC vs. Technical Debt

Testing
Challenges & Takeaways

Takeaways

- Coverage metrics
  - *just an indicator, they do not "sign off" on software quality*
  - *provide guidance: areas that need attention*
- Tests are code
  - *update tests as code evolves (e.g., every time you fix a bug, write a test for it)*
  - *keep tests under version control*
  - *run often*
  - *should be of production quality*
- Look at the big picture
  - *consider technical debt, code smells, etc.*

Takeaways

- High coverage ⇒ well-exercised codebase ⇏ freedom from defects
- Coverage is an imperfect proxy metric for test quality
  - *measures how much code is executed by tests, not how well they test the code*
  - *does not measure the quality and importance of the test scenarios*
- Challenge: Not all code is equal
- Challenge: Flaky tests
- Challenge: Contextual sensitivity
  - *the more circumstances need to combine in order for a bug to manifest, the harder it is to anticipate and reveal them in testing (e.g., concurrency issues)*

Some Famous "Corner-Case" Bugs

- Spring4Shell (2022)
- Dirty Pipe (2022)
- Log4Shell (2021)
- SolarWinds Orion Platform (2020)
- WannaCry Ransomware Attack (2017)
- Cloudbleed (2017)
- Heartbleed (2014)
- Knight Capital Group Trading Disaster (2012)
- DNS Cache Poisoning (2008)
- Y2K Bug (Year 2000)
- Mars Climate Orbiter Disintegration (1999)
- Ariane 5 Rocket Flight 501 (1996)
- Patriot Missile (1991):
- Therac-25 Radiation Therapy Machine (1980s)

Some Famous "Corner-Case" Bugs

- Spring4Shell (2022)
- Dirty Pipe (2022)
- Log4Shell (2021)
- SolarWinds Orion Platform (2020)
- WannaCry Ransomware Attack (2017)
- Cloudbleed (2017)
- Heartbleed (2014)

- Knight Capital Group Trading Disaster (2012)
- DNS Cache Poisoning (2008)
- Y2K Bug (Year 2000)
- Mars Climate Orbiter Disintegration (1999)
- Ariane 5 Rocket Flight 501 (1996)
- Patriot Missile (1991):
- Therac-25 Radiation Therapy Machine (1980s)

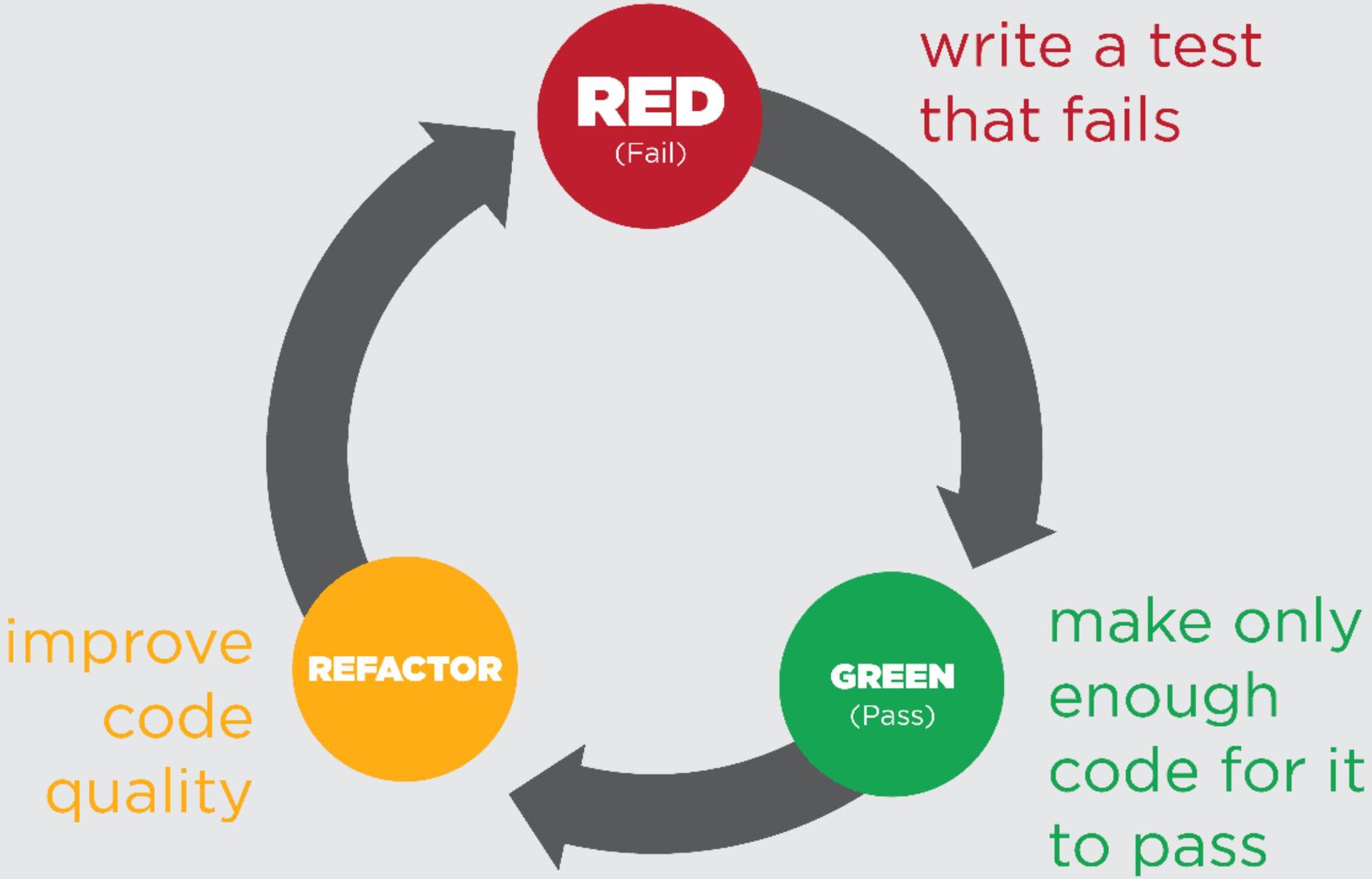
Smartphone Platform: Heterogeneity Challenges to Testing

- Devices
  - *device-specific or configuration-specific bugs*
- Integration with external systems
  - *you depend on the network !*
  - *unexpected behavior can cause mobile app to misbehave*
- Users
  - *watch your input validation*
- Sensors are finicky
- Interaction with system features

Testing Challenges: Non-Functional Requirements

- performance, security, usability, compatibility, etc.
- test coverage numbers do not capture these directly
  - *yet they can significantly impact user experience*
- static analysis tools
- performance and load testing
- security testing (both automated and manual)
- exploratory testing

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())
    }

    ...
```

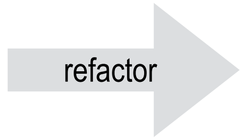
```
class Calculator {

    // 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()
    }
}
```

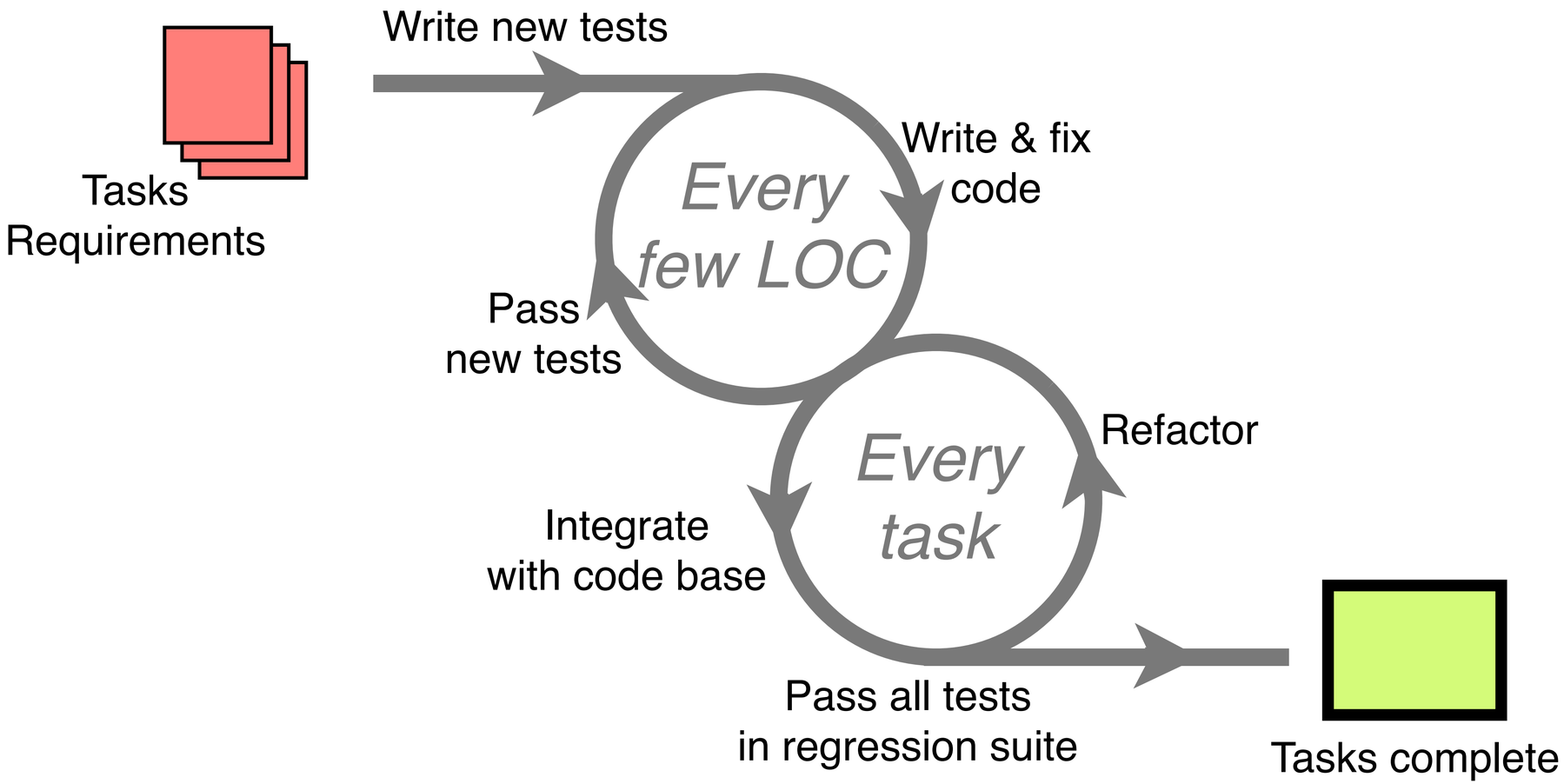


```
interface CalculatorUI {
    fun display(result: String)
}

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/

Behavior-Driven Development (BDD)

Formalizing user stories

```
Scenario: Guess a word
    Given that the Wordmaker starts the game
     When the Wordmaker has started the game
     Then the Wordmaker waits for the player to join
```

Formalizing user stories: Steps

```
Given <context>
When <event>
Then <expected outcome>
```

Given user is not logged in...

Given player has at least 1 life left...

Formalizing user stories: Steps

```
Given <context>
When <event>
Then <expected outcome>
```

When user edits profile...

When player steps on bomb...

Formalizing user stories: Steps

```
Given <context>

When <event>

Then <expected outcome>
```

Then user gets an error message

Then number of lives decreases by 1

Formalizing user stories: Scenarios

Scenario: All done

Given I am out shopping

Given I have eggs

Given I have milk

Given I have butter

When I check my list

Then I smile because I don't need anything

Formalizing user stories: Features

```
Feature: User Login
  Scenario: Successful login
    Given user navigates to polyfood.ch website
    When user logs in using Username "USER" and Password "goodPASSWORD"
    Then login should be successful
  Scenario: Failed login
    Given user navigates to polyfood.ch website
    When user logs in using Username "USER" and Password "badPASSWORD"
    Then error message should be thrown
```

Formalizing user stories: Features

```
Feature: Multiple site support
  Only blog owners can post to a blog, except administrators, who can post to all blogs.

  Background:
    Given a global administrator named "Greg"
    And a blog named "Greg's anti-tax rants"
    And a customer named "Dr. Bill"
    And a blog named "Expensive Therapy" owned by "Dr. Bill"

  Scenario: Dr. Bill posts to his own blog
    Given I am logged in as Dr. Bill
    When I try to post to "Expensive Therapy"
    Then I should see "Your article was published."

  Scenario: Dr. Bill tries to post to somebody else's blog, and fails
    Given I am logged in as Dr. Bill
    When I try to post to "Greg's anti-tax rants"
    Then I should see "Hey! That's not your blog!"

  Scenario: Greg posts to a client's blog
    Given I am logged in as Greg
    When I try to post to "Expensive Therapy"
    Then I should see "Your article was published."
```

https://github.com/1c-syntax/turbo-gherkin-parser/blob/master/src/test/resources/correct/example3.feature

Scenarios → Acceptance Tests

```
Feature: User Login

  Scenario: Successful Login
    Given user navigates to polyfood.ch website
    When user logs in using Username as "jane_doe" and Password "password123"
    And clicks the Submit button
    Then Home page should be displayed
    And login should be successful
```

login.feature

```
class LoginSteps {
    @Given("user navigates to polyfood.ch website")
    fun navigatePage() {
        println("Cucumber executed Given statement")
        // Add actual navigation logic
    }

    @When("user logs in using Username as {string} and Password {string}")
    fun login(username: String, password: String) {
        println("Username is: $username")
        println("Password is: $password")
        // Add login logic here
    }

    @When("clicks the Submit button")
    fun clickTheSubmitButton() {
        println("Executing When statement")
        // Add button click logic here
    }

    @Then("Home page should be displayed")
    fun validatePage() {
        println("Executing 1st Then statement")
        // Add home page validation logic
    }

    @Then("login should be successful")
    fun validateLoginSuccess() {
        println("Executing 2nd Then statement")
        // Add login success validation logic
    }
}
```

LoginSteps.kt

Piecing it all together

```
Feature: User Login

  Scenario: Successful Login
    Given user navigates to polyfood.ch website
    When user logs in using Username as "jane_doe" and Password "password123"
    And clicks the Submit button
    Then Home page should be displayed
    And login should be successful
```

login.feature

```
class LoginSteps {
    @Given("user navigates to polyfood.ch website")
    fun navigatePage() {
        println("Cucumber executed Given statement")
        // Add actual navigation logic
    }

    @When("user logs in using Username as {string} and Password {string}")
    fun login(username: String, password: String) {
        println("Username is: $username")
        println("Password is: $password")
        // Add login logic here
    }

    @When("clicks the Submit button")
    fun clickTheSubmitButton() {
        println("Executing When statement")
        // Add button click logic here
    }

    @Then("Home page should be displayed")
    fun validatePage() {
        println("Executing 1st Then statement")
        // Add home page validation logic
    }

    @Then("login should be successful")
    fun validateLoginSuccess() {
        println("Executing 2nd Then statement")
        // Add login success validation logic
    }
}
```

LoginSteps.kt

- Keep features independent
- Ideally keep scenarios independent

1

As a vacation traveler,
I want to be able to get a refund anytime before my arrival,
so that I can get my money back in the event of an emergency.

`user story`

2

```
Feature: Vacation Traveler Refund

  Scenario Outline: Refund reservation
    Given the user has a reservation id "<id>"
    When the user clicks on the "refund" button
    Then a refund order is made for the reservation id "<id>"

  Examples:
    | id    |
    | 12345 |
    | 67890 |
```

`refund.feature`

3

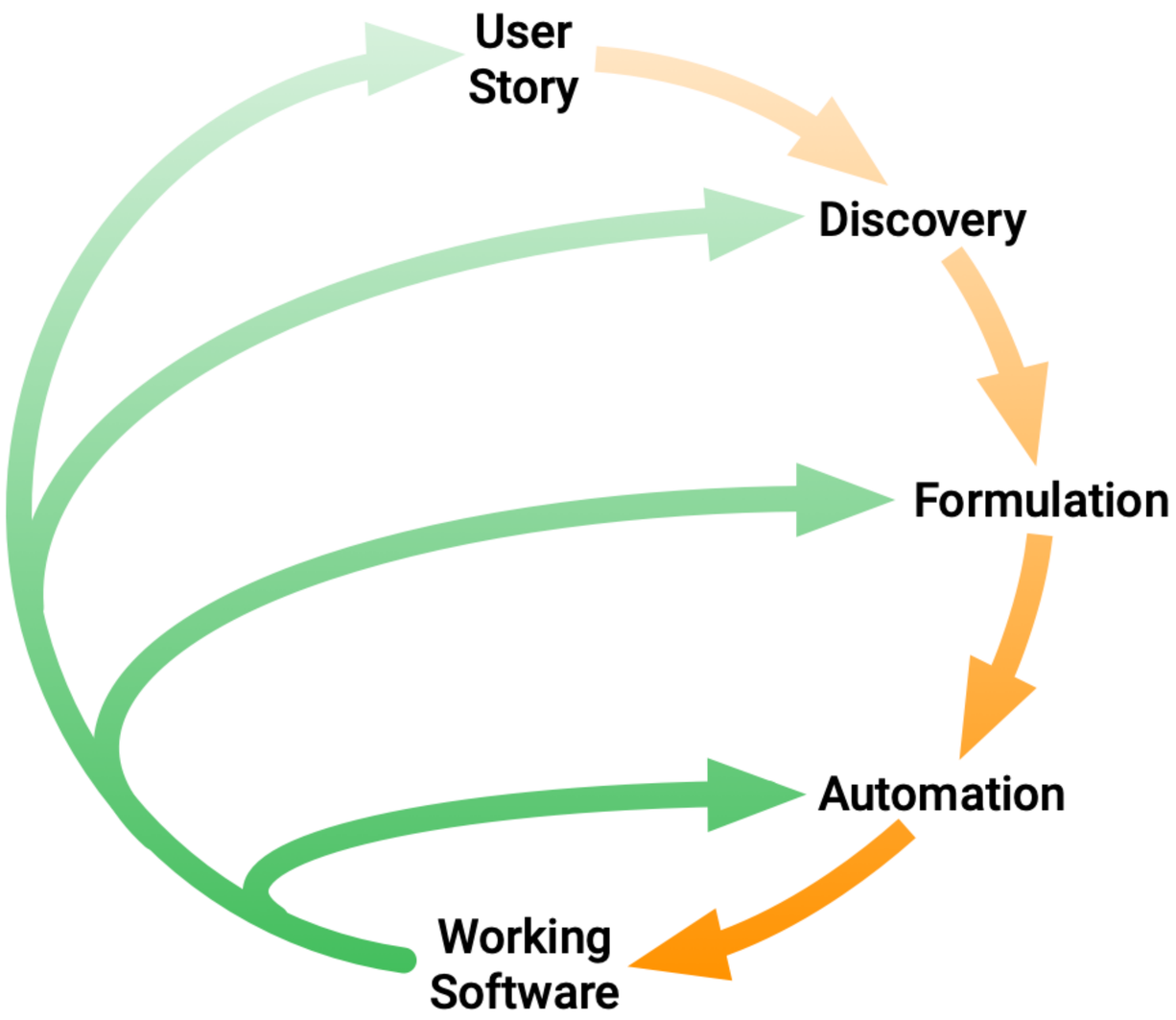
```
class RefundSteps {

    private lateinit var reservationId: String

    @Given("the user has a reservation id {string}")
    fun theUserHasAReservationId(id: String) {
        reservationId = id
        // ... simulate retrieving the reservation by id ...
        println("Retrieved reservation with id: $reservationId")
    }

    @When("the user clicks on the {string} button")
    fun theUserClicksOnTheButton(buttonName: String) {
        // ... simulate clicking on the refund button ...
        println("User clicked on the $buttonName button")
    }

    @Then("a refund order is made for the reservation id {string}")
    fun aRefundOrderIsMadeForTheReservationId(id: String) {
        // ... simulate making a refund order for the reservation ...
        assert(reservationId == id)
        // ... more asserts to verify the actual refund logic check ...
        println("A refund order has been made for reservation id: $id")
    }
}
```

`RefundSteps.kt`

4

Implementation

Principles of BDD



Outline

- Recap of Testing (CS-214)
- Cost of Bugs
  - *the later you eliminate the defect, the more expensive it is*
- How well can we test?
  - *"Testing can be used to show the presence of bugs, never their absence!"*
  - *Measure coverage: statement / branch / path*
- Coverage Metrics in Practice
- Test-Driven Development (TDD)
- Behavior-Driven Development (BDD)