CS-311: Testing Fundamentals

Prof. George Candea
School of Computer & Communication Sciences

Outline

- Recap of Testing
- Cost of Bugs
- How well can we test?
- Coverage Metrics
- Test-Driven Development (TDD) — *online*
- Behavior-Driven Development (BDD) — *online*

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)*
- End-to-end tests
  - *user journey from start to finish (e2e is a system test, but system is not necessarily e2e ...)*
- Acceptance tests
  - *testing the product in real-world scenarios to ensure it meets stakeholders' requirements*

```kotlin
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) {
        ...
    }
}
```

```kotlin
@Test
fun calculateTotalReturnsCorrectTotal() {
    val items = listOf(
        CartItem("Pizza", 10.0, 2),
        CartItem("Burger", 5.0, 3)
    )
    val order = Order(items)

    val total = order.calculateTotal()

    assertEquals(35.0, total, 0.001)
}
```

What kind of test is this?

```kotlin
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) {
        ...
    }
}
```

```kotlin
@Test
fun calculateTotalReturnsCorrectTotal() {
    val items = listOf(
        CartItem("Pizza", 10.0, 2),
        CartItem("Burger", 5.0, 3)
    )
    val order = Order(items)

    val total = order.calculateTotal()

    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*

```kotlin
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: String): 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)
```

```kotlin
@Test
fun successfulOrderPlacement() {
  val order = Order(
      listOf(CartItem("Pizza", 10.0, 2),
             CartItem("Burger", 5.0, 3))
  )    // Create an order with 2 pizzas (10.0 each) and 3 burgers (5.0 each)

  val orderRepo = InMemoryOrderRepository() // in-memory OrderRepository for testing
  val paymentProcessor = SandboxPaymentProcessor() // sandbox API simulates payment without charging

  val service = OrderService(orderRepo, paymentProcessor) // the order service under test
  val result = service.placeOrder(order)

  val saved = orderRepo.findById(result.orderId)!!  // retrieve the saved order from the repository

  assertEquals(OrderStatus.CONFIRMED, saved.status()) // verify the order status
  assertEquals(35.0, saved.calculateTotal(), 0.001) // verify the total price
  assertNotNull(saved.transactionId()) // verify that order has a proper txn ID
}
```

What kind of test is this?

```kotlin
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: String): 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)
```

```kotlin
@Test
fun successfulOrderPlacement() {
  val order = Order(
      listOf(CartItem("Pizza", 10.0, 2),
             CartItem("Burger", 5.0, 3))
  )    // Create an order with 2 pizzas (10.0 each) and 3 burgers (5.0 each)

  val orderRepo = InMemoryOrderRepository() // in-memory OrderRepository for testing
  val paymentProcessor = SandboxPaymentProcessor() // sandbox API simulates payment without charging

  val service = OrderService(orderRepo, paymentProcessor) // the order service under test
  val result = service.placeOrder(order)

  val saved = orderRepo.findById(result.orderId)!!  // retrieve the saved order from the repository

  assertEquals(OrderStatus.CONFIRMED, saved.status()) // verify the order status
  assertEquals(35.0, saved.calculateTotal(), 0.001) // verify the total price
  assertNotNull(saved.transactionId()) // verify that order has a proper txn ID
}
```

Integration test

```kotlin
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
            }
        }
    }
}
```

What kind of test is this?

```kotlin
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 & E2E 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 & E2E test

Notes for Compose ...

- add Modifier.testTag("...") to the key Composables in the UI code... don't rely only on text when multiple items share labels
- use a clear confirmation signal after checkout (toast/snackbar/dialog text like “Order confirmed” or a view with tag orderConfirmed)
- keep the data deterministic in test (e.g., seed a fake repo/payment service in MainActivity when BuildConfig.DEBUG && isTest)

```
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"
```

What kind of test is this?

```gherkin
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

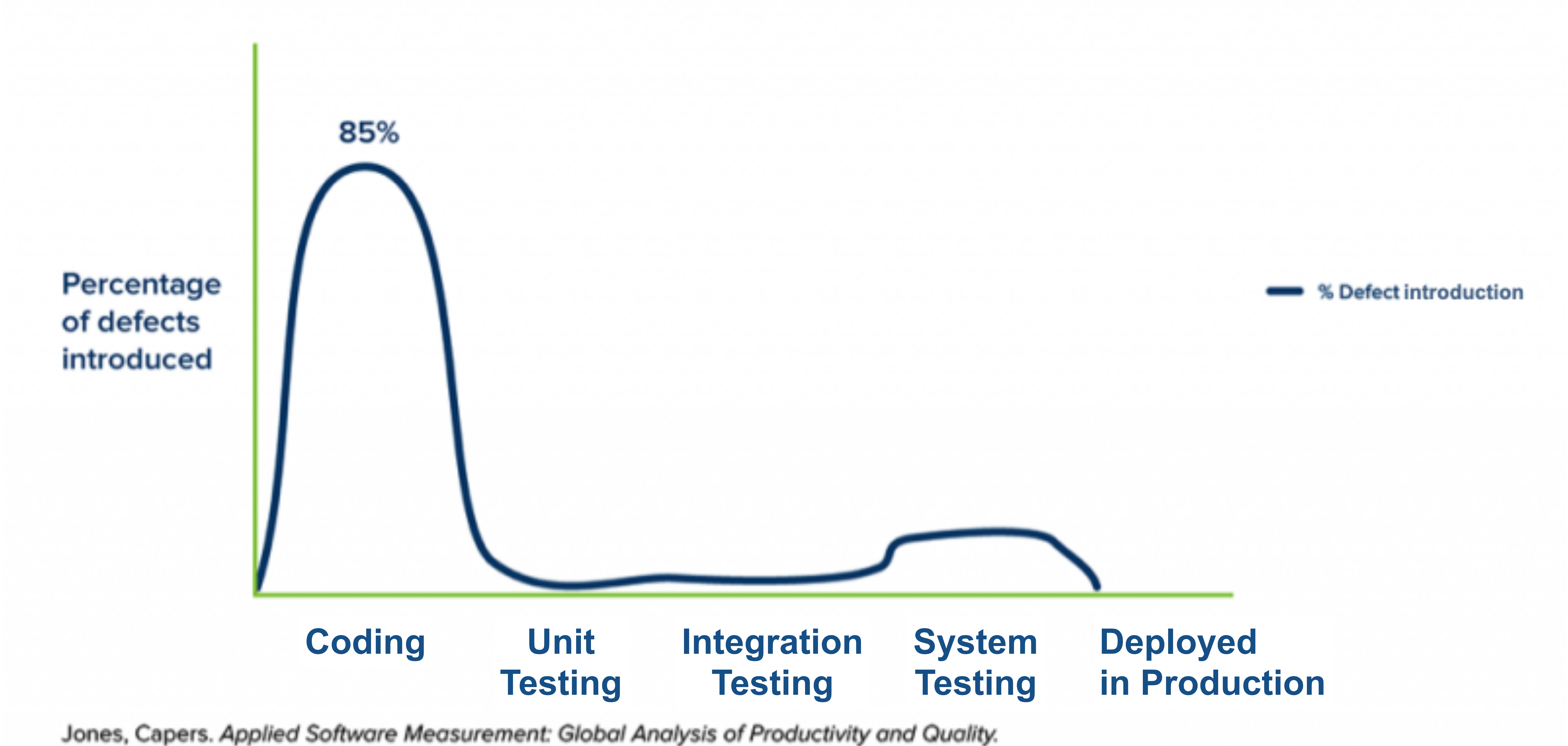
Testing: Different Levels *(recap from CS-214)*

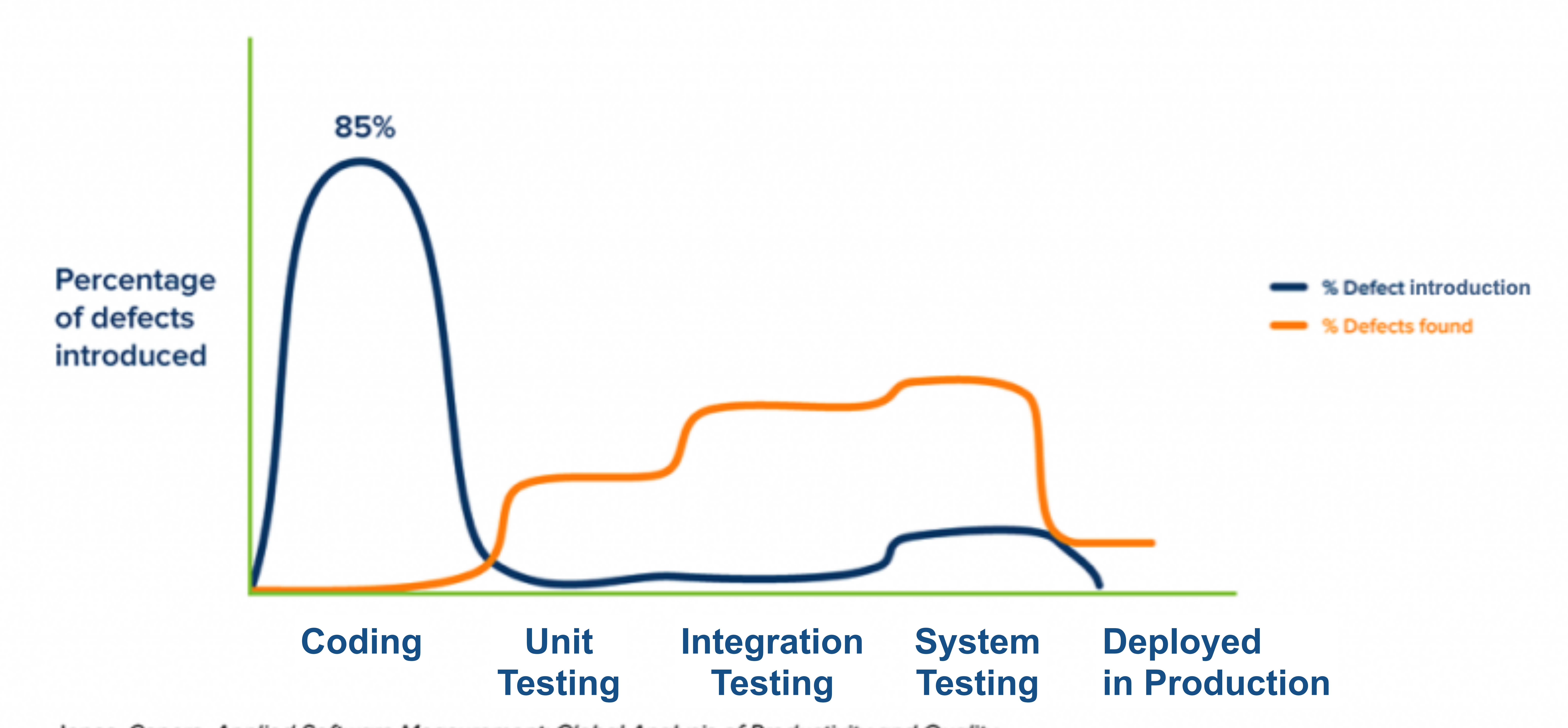
- Unit tests
  - *individual components or functions (e.g., a user input validation function of the app)*
- Integration tests
  - *test combinations of components (e.g., login interface + backend database)*
- System tests
  - *the complete and integrated software (e.g., testing the entire app's functionality)*
- End-to-end tests
  - *user journey from start to finish (e2e is a system test, but system is not necessarily e2e ...)*
- Acceptance tests
  - *testing the product in real-world scenarios to ensure it meets user requirements*

Outline

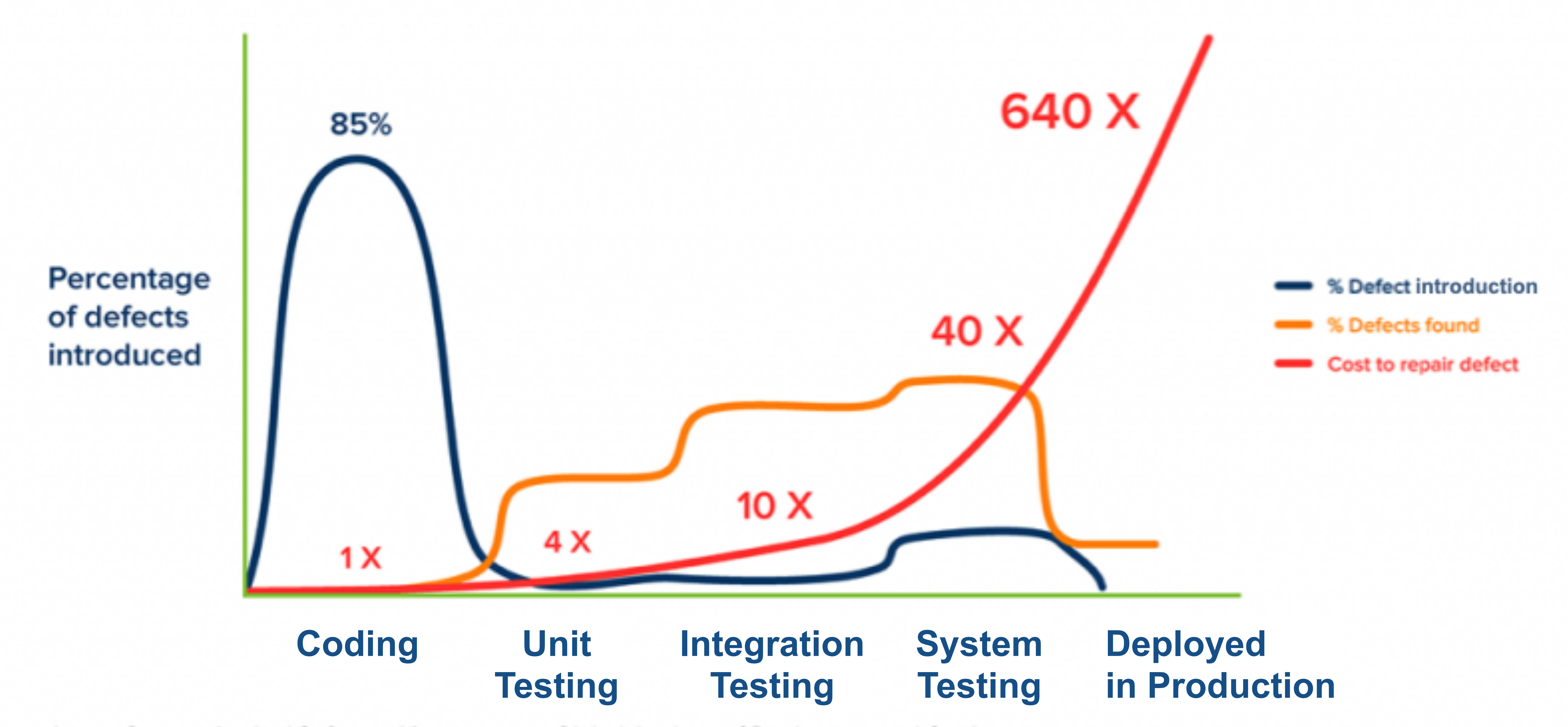
- Recap of Testing
- Cost of Bugs
- How well can we test?
- Coverage Metrics
- Test-Driven Development (TDD) — *online*
- Behavior-Driven Development (BDD) — *online*

Cost of Bugs

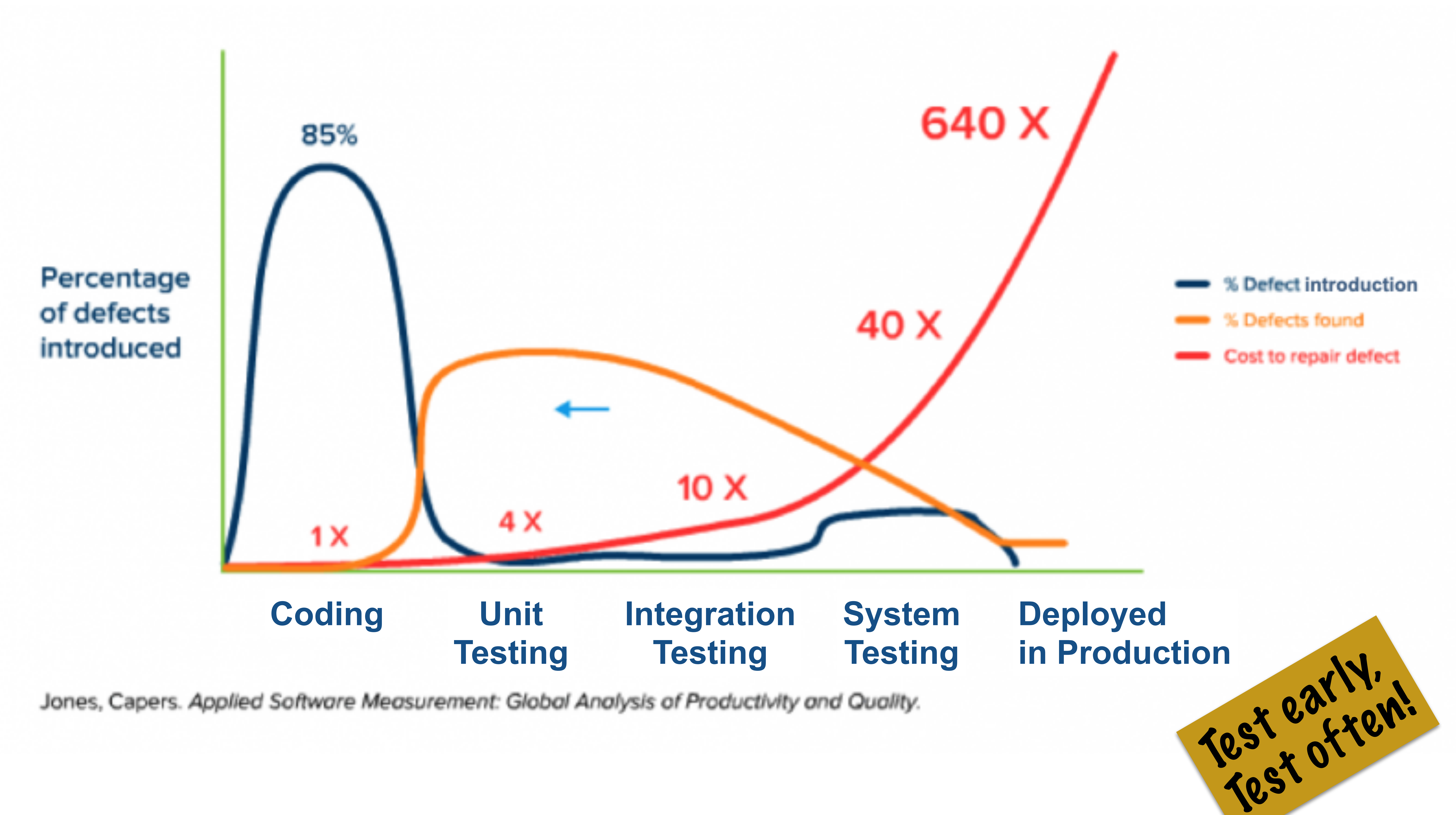

85%
Percentage of defects introduced
% Defect introduction
Coding
Unit Testing
Integration Testing
System Testing
Deployed in Production
Jones, Capers. Applied Software Measurement: Global Analysis of Productivity and Quality.
https://www.stickyminds.com/article/shift-left-approach-software-testing

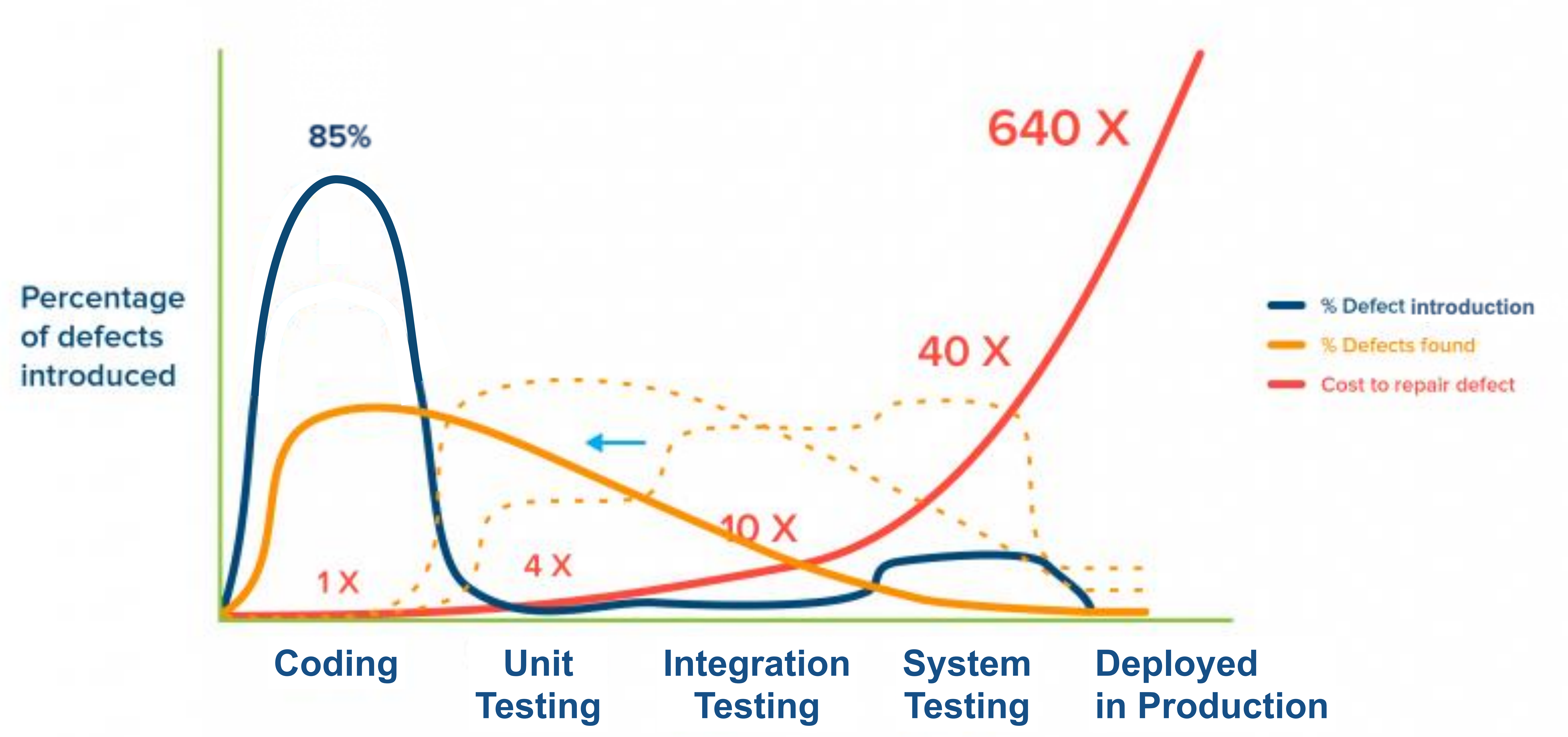


https://www.stickyminds.com/article/shift-left-approach-software-testing

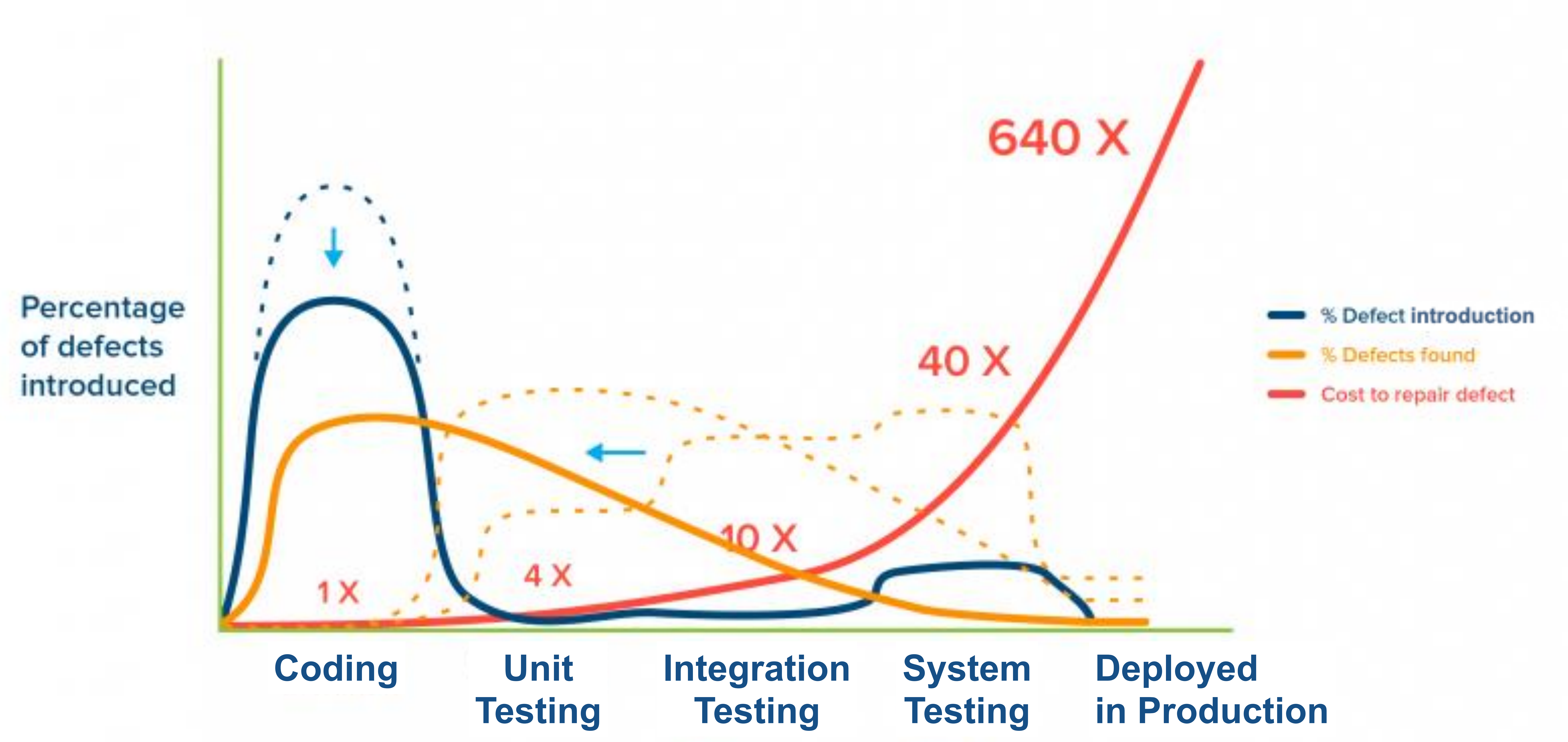


https://www.stickyminds.com/article/shift-left-approach-software-testing


85%
640 X
40 X
10 X
4 X
1 X
Percentage of defects introduced
% Defect introduction
% Defects found
Cost to repair defect
Coding
Unit Testing
Integration Testing
System Testing
Deployed in Production
Jones, Capers. Applied Software Measurement: Global Analysis of Productivity and Quality.
Test early, Test often!
https://www.stickyminds.com/article/shift-left-approach-software-testing



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



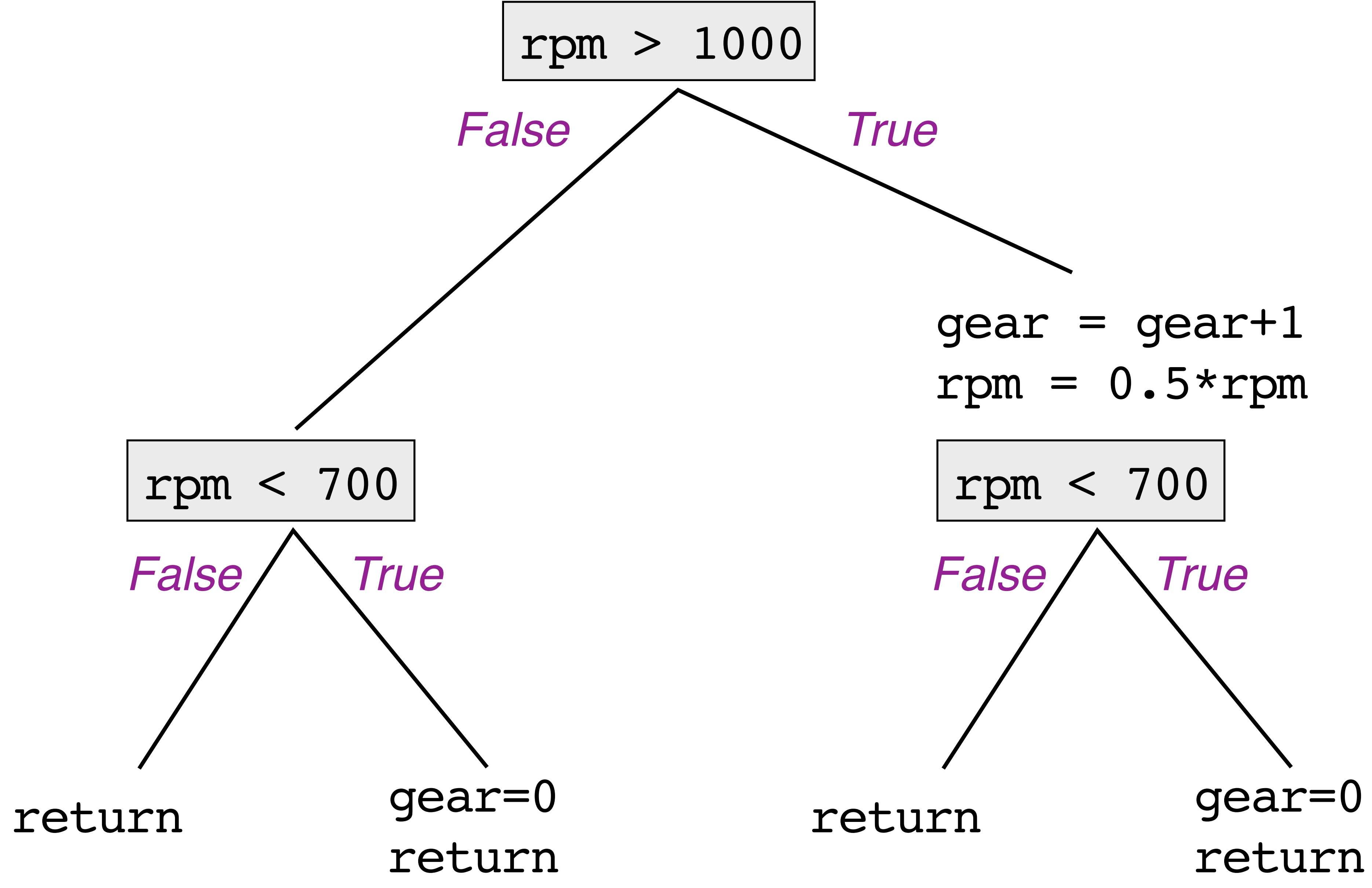
Jones, Capers. *Applied Software Measurement: Global Analysis of Productivity and Quality.*

Outline

- Recap of Testing
- Cost of Bugs
- How well can we test?
- Coverage Metrics
- Test-Driven Development (TDD) — *online*
- Behavior-Driven Development (BDD) — *online*

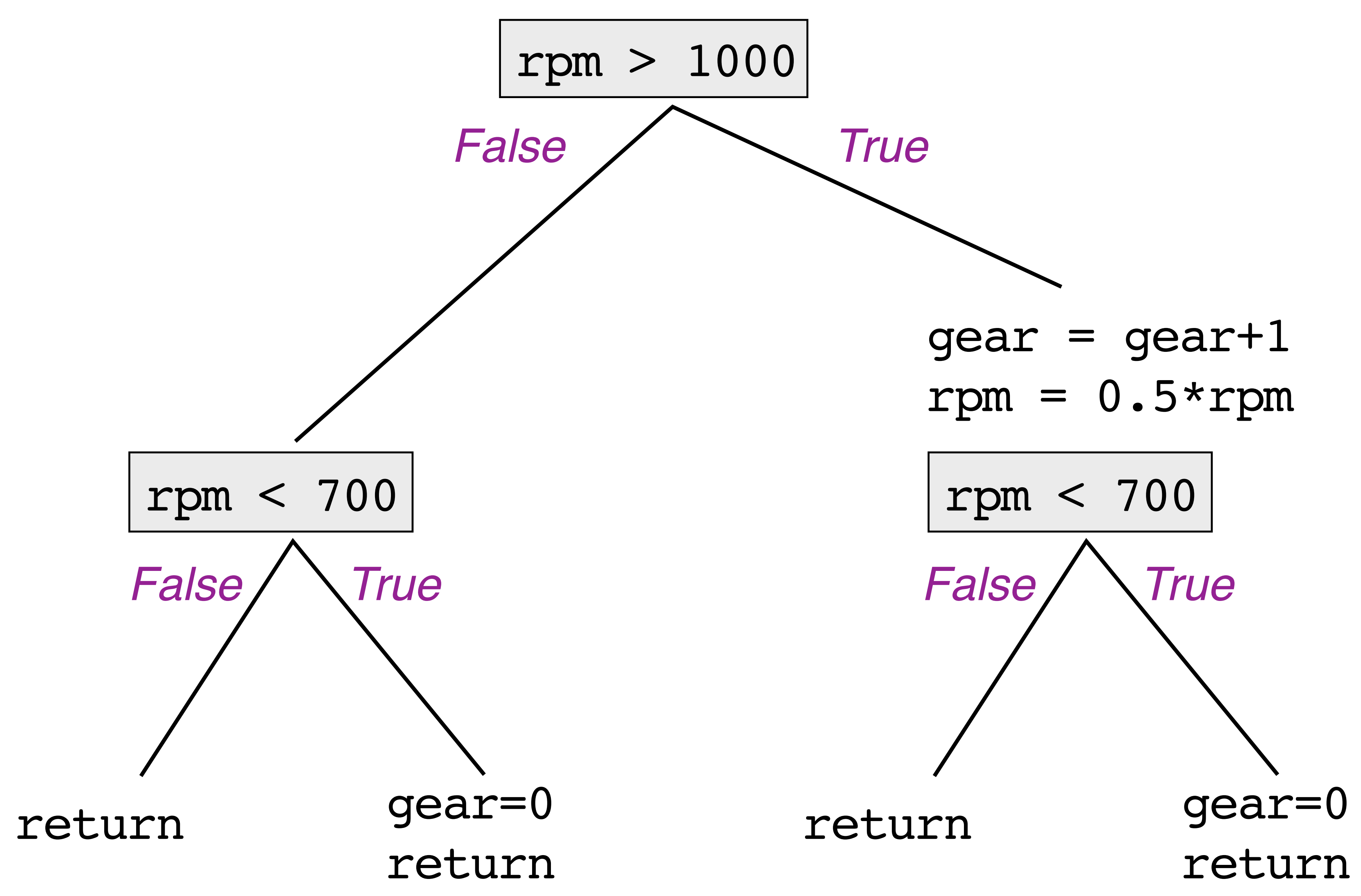
How well can we test?

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

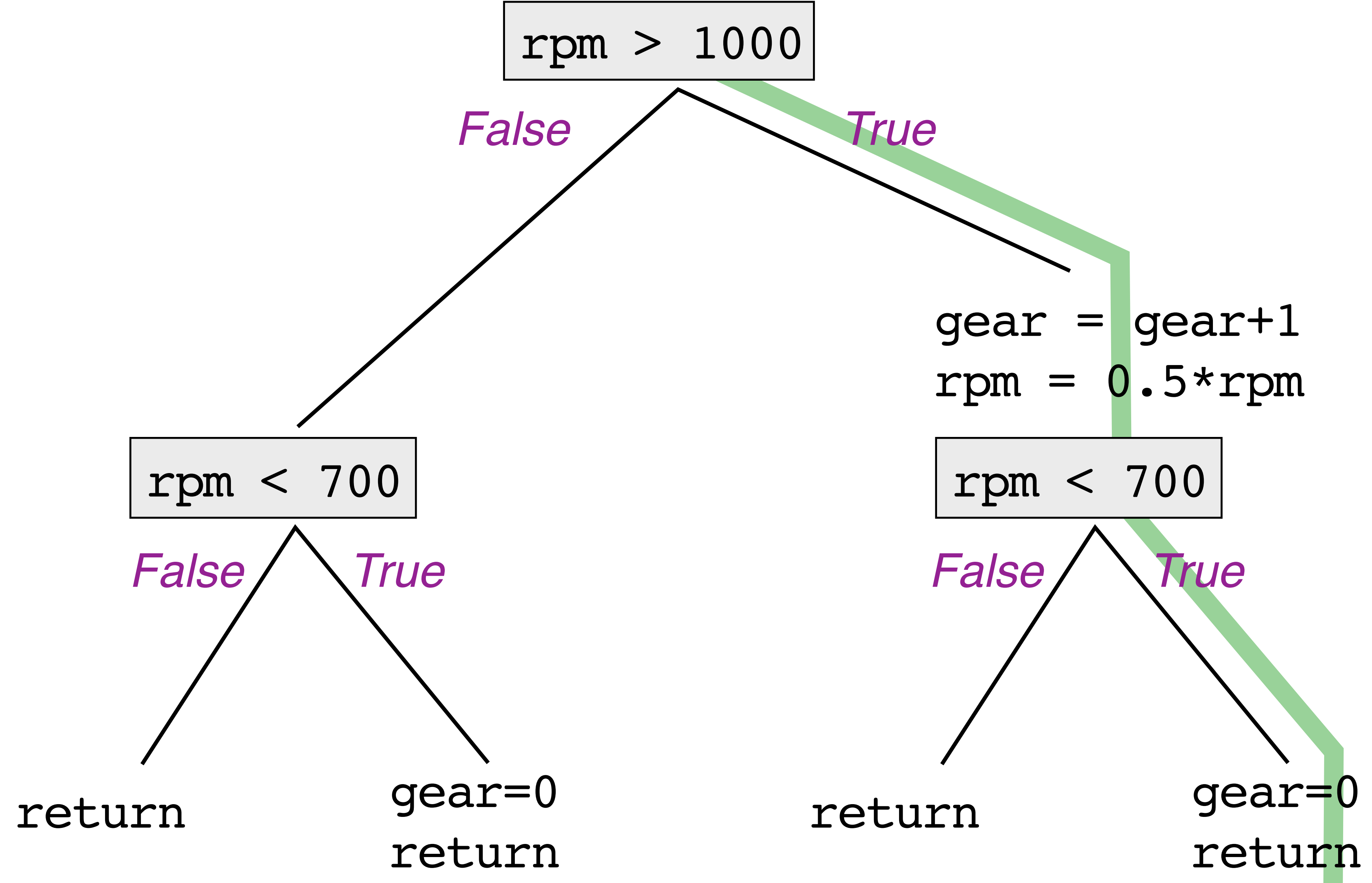


rpm=1200

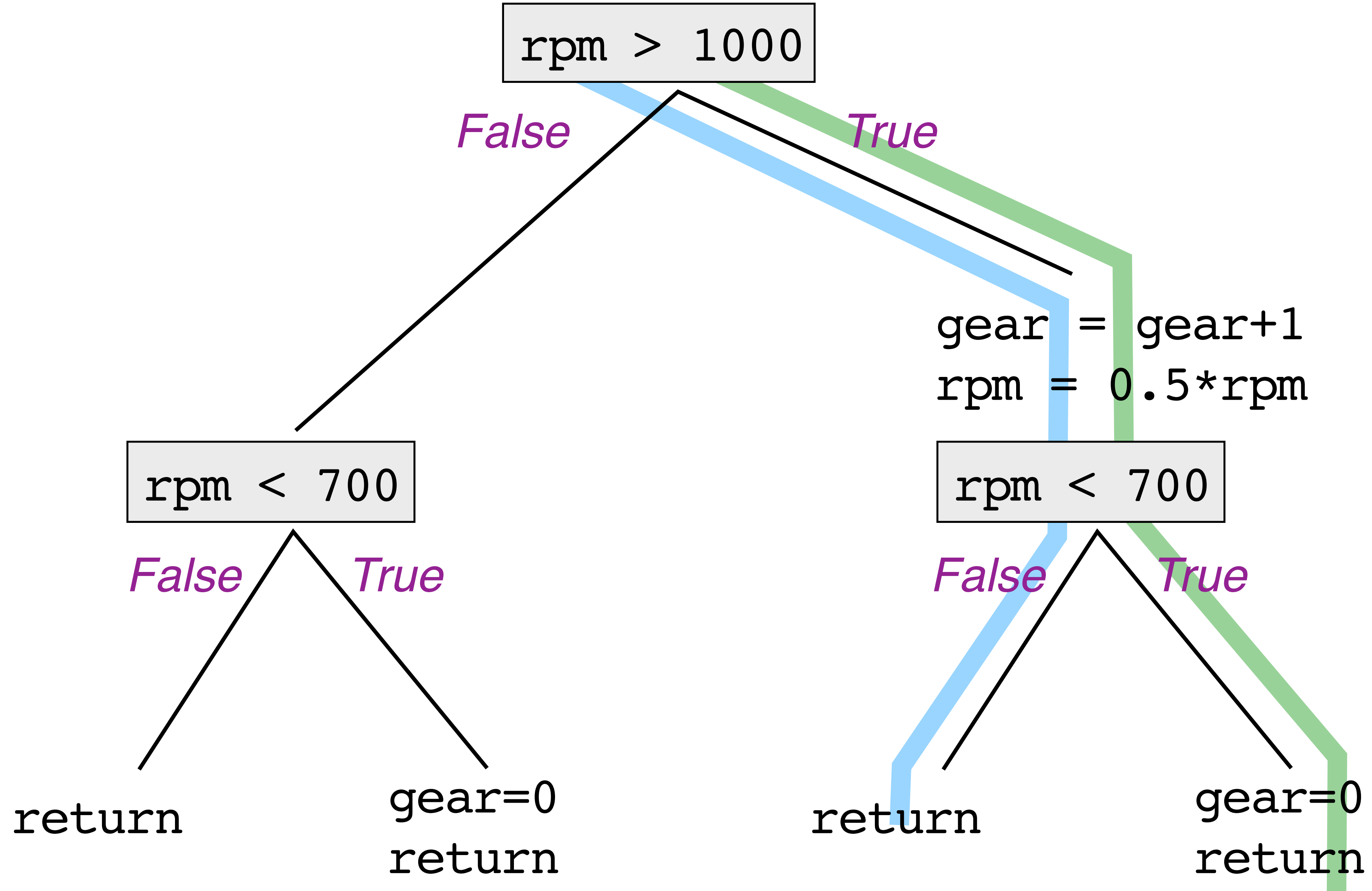
```
autoShift (int rpm)
   if (rpm > 1000)
      gear = gear+1
      rpm = 0.5*rpm
   if (rpm < 700)
      gear=0
   return
```



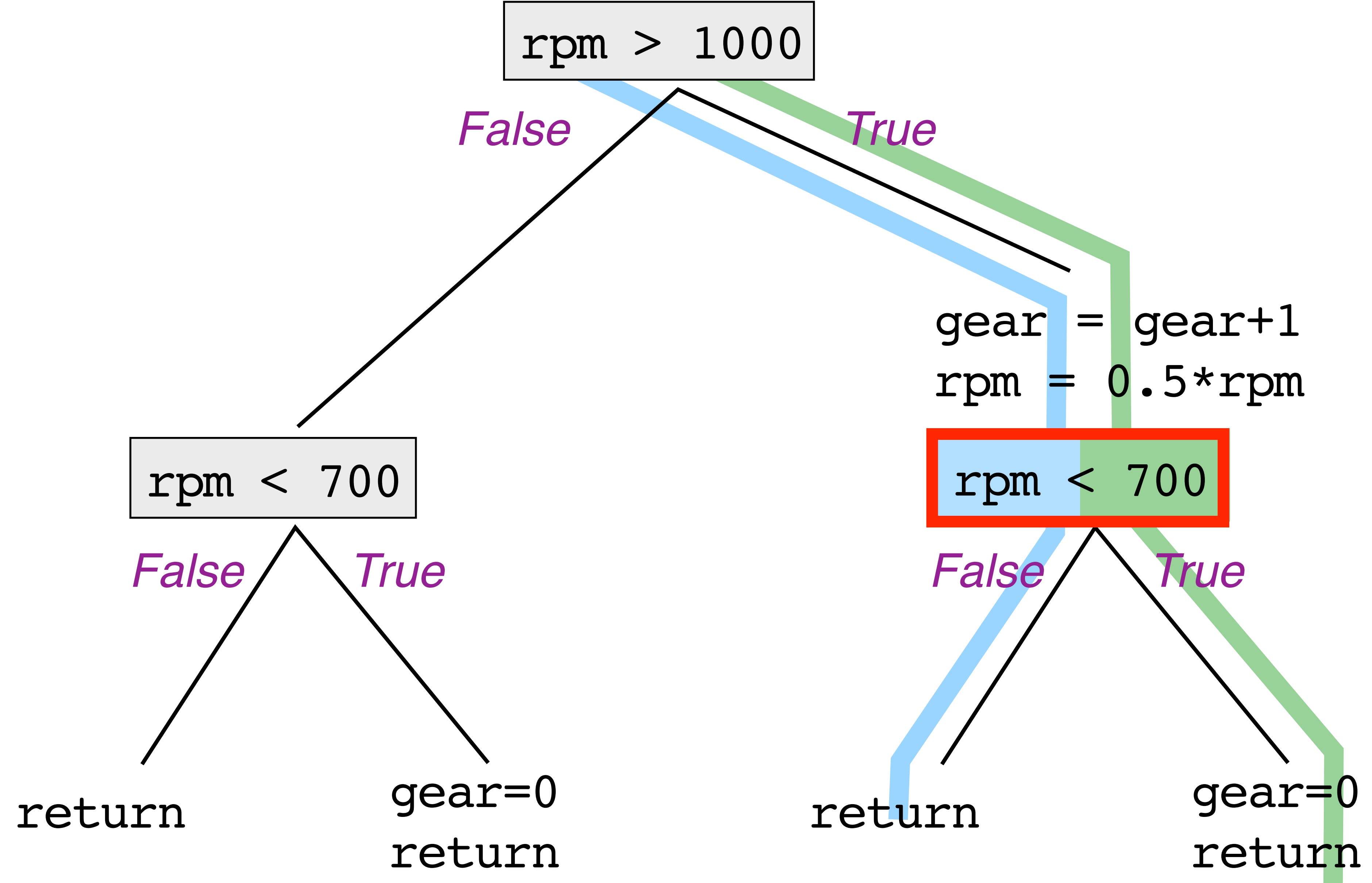
```
autoShift (int rpm)
    if (rpm > 1000)
        gear = gear+1
        rpm = 0.5*rpm
    if (rpm < 700)
        gear=0
    return
```



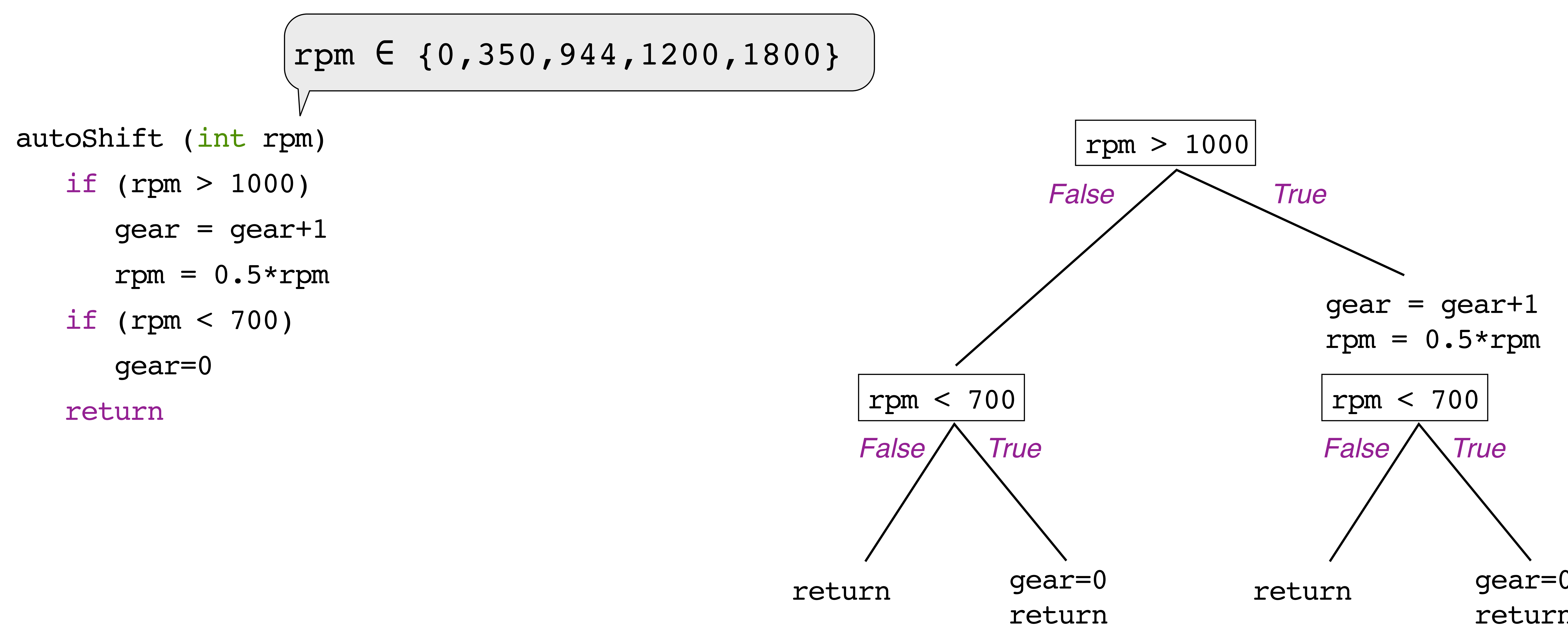
```
autoShift (int rpm)
   if (rpm > 1000)
      gear = gear+1
      rpm = 0.5*rpm
   if (rpm < 700)
      gear=0
   return
```



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

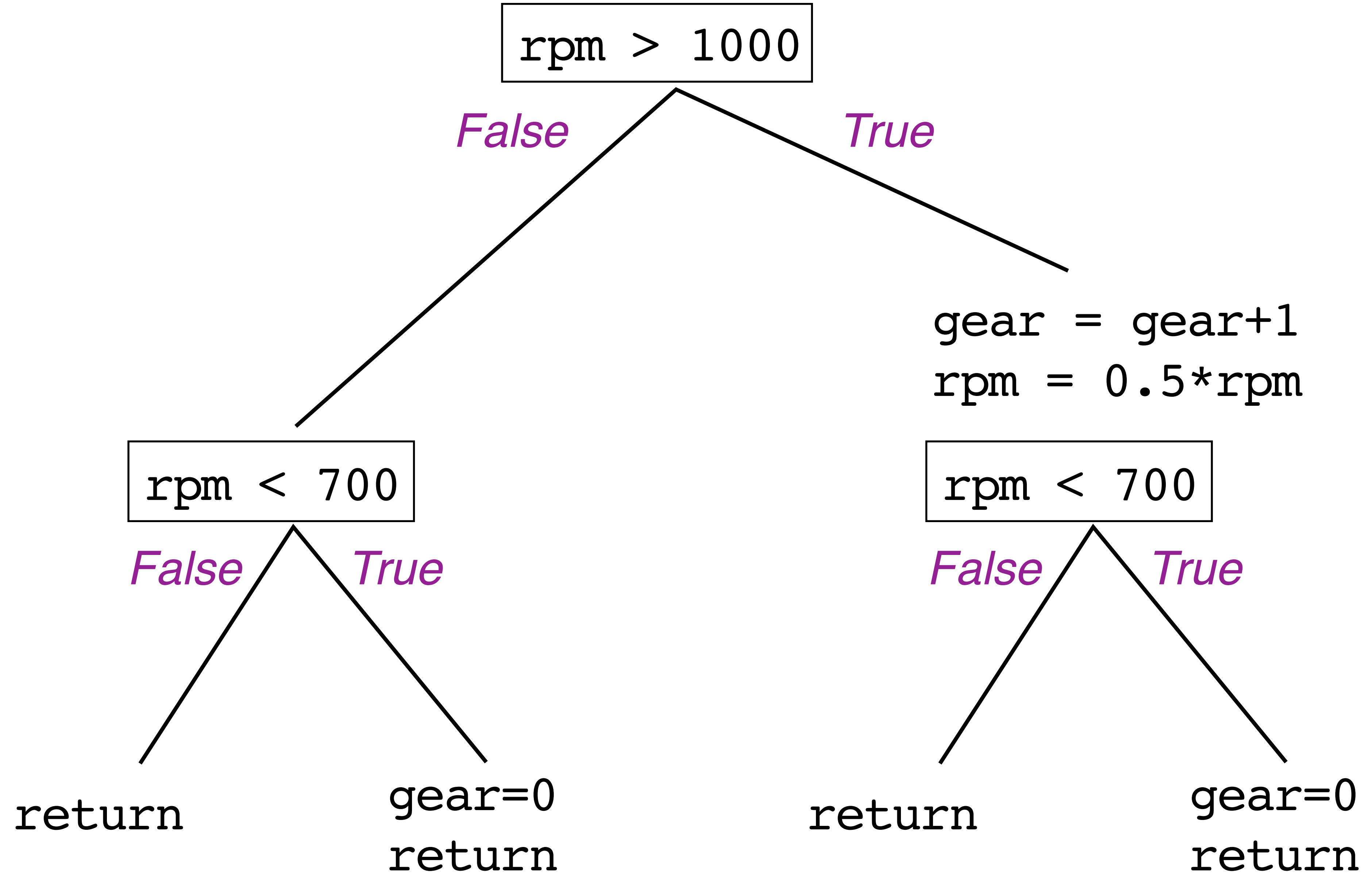


Software Testing



Software Testing

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



paths ≃ $2^{\text{program size}}$

How many possible execution paths (behaviors) ?

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

How many possible execution paths (behaviors) ?

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

2

How many possible execution paths (behaviors) ?

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

2

```
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;
}
```

How many possible execution paths (behaviors) ?

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

2

```
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;
}
```

3

How many possible execution paths (behaviors) ?

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

2

```
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;
}
```

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;
}
```

How many possible execution paths (behaviors) ?

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

2

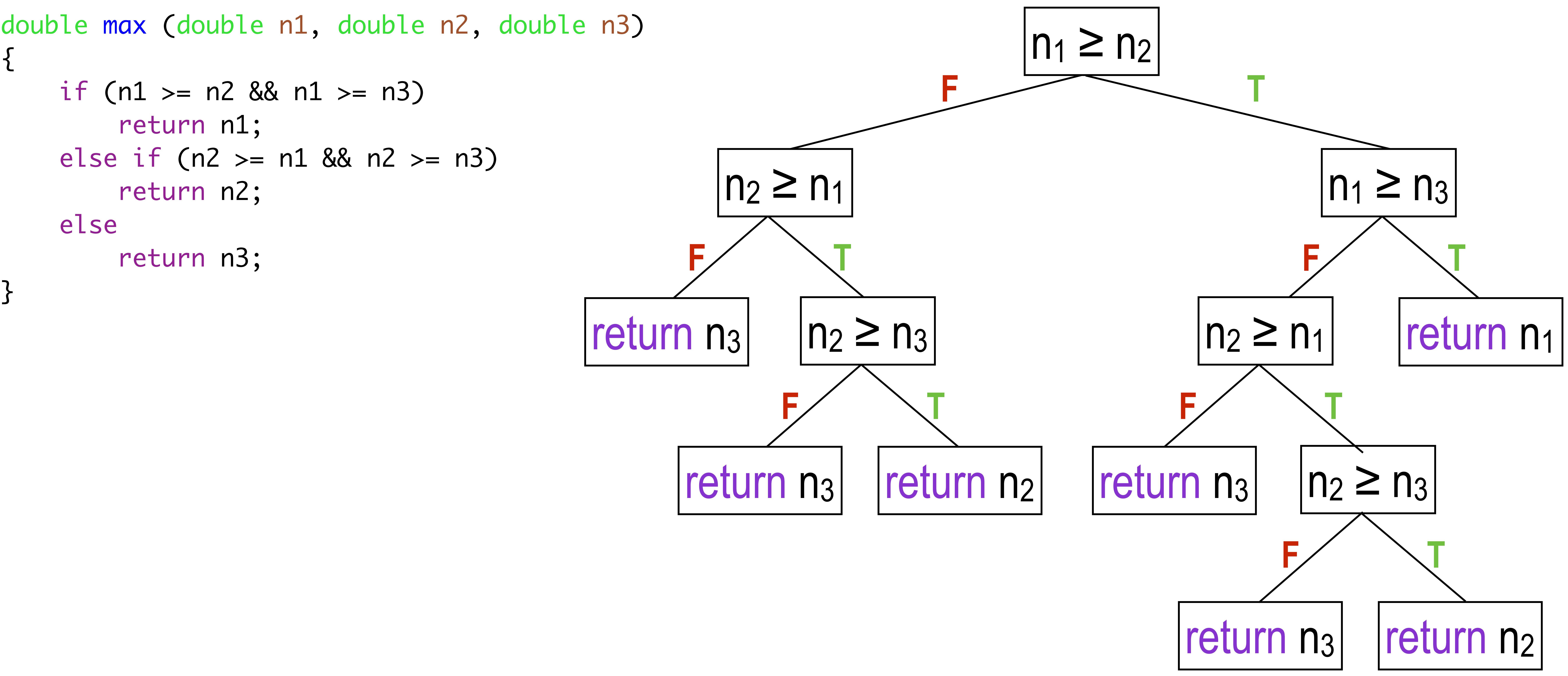
```
double max (double n1, double n2, double n3)
{
    if (n1 >= n2 & n1 >= n3)
        return n1;
    else if (n2 >= n1 & n2 >= n3)
        return n2;
    else
        return n3;
}
```

3

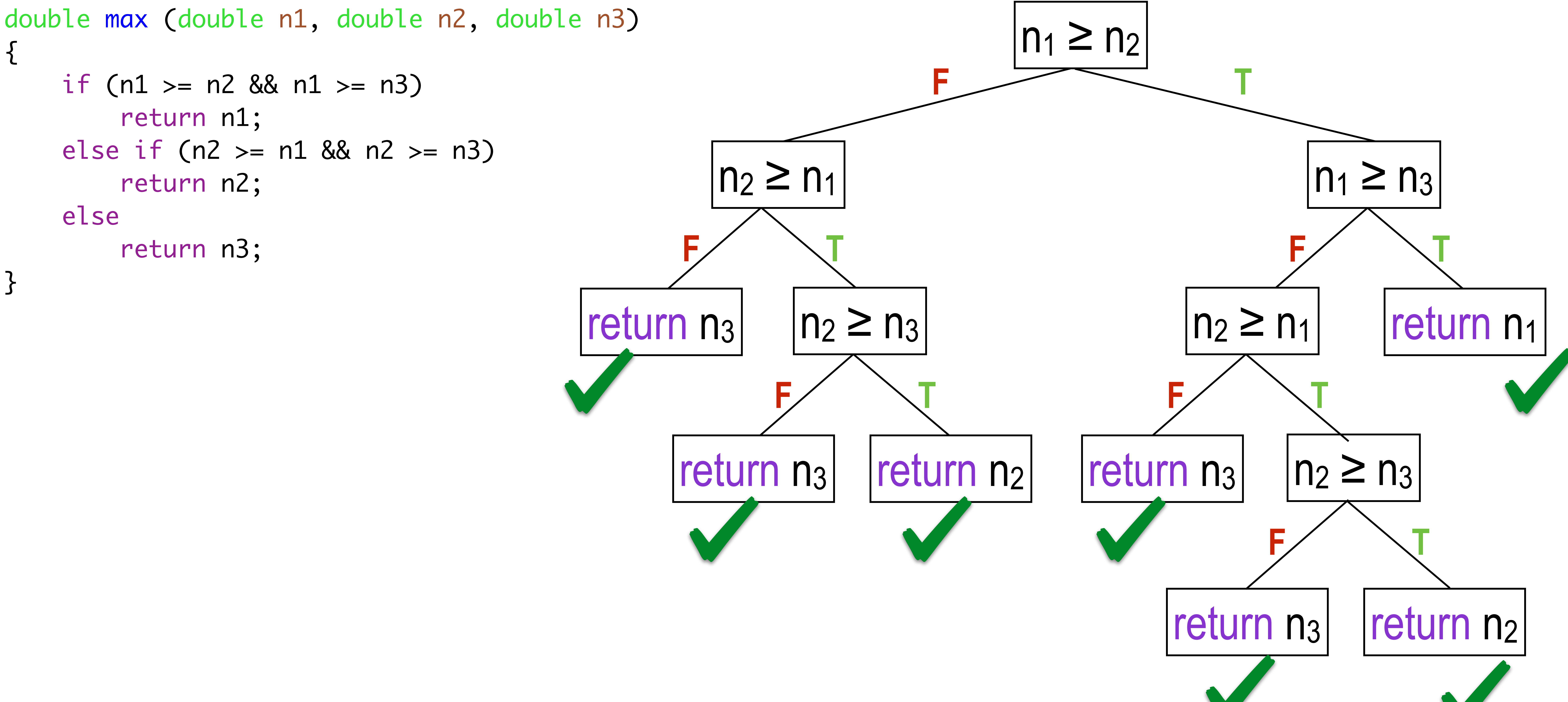
```
double max (double n1, double n2, double n3)
{
    if (n1 >= n2 && n1 >= n3)
        return n1;
    else if (n2 >= n1 && n2 >= n3)
        return n2;
    else
        return n3;
}
```

short-circuit evaluation

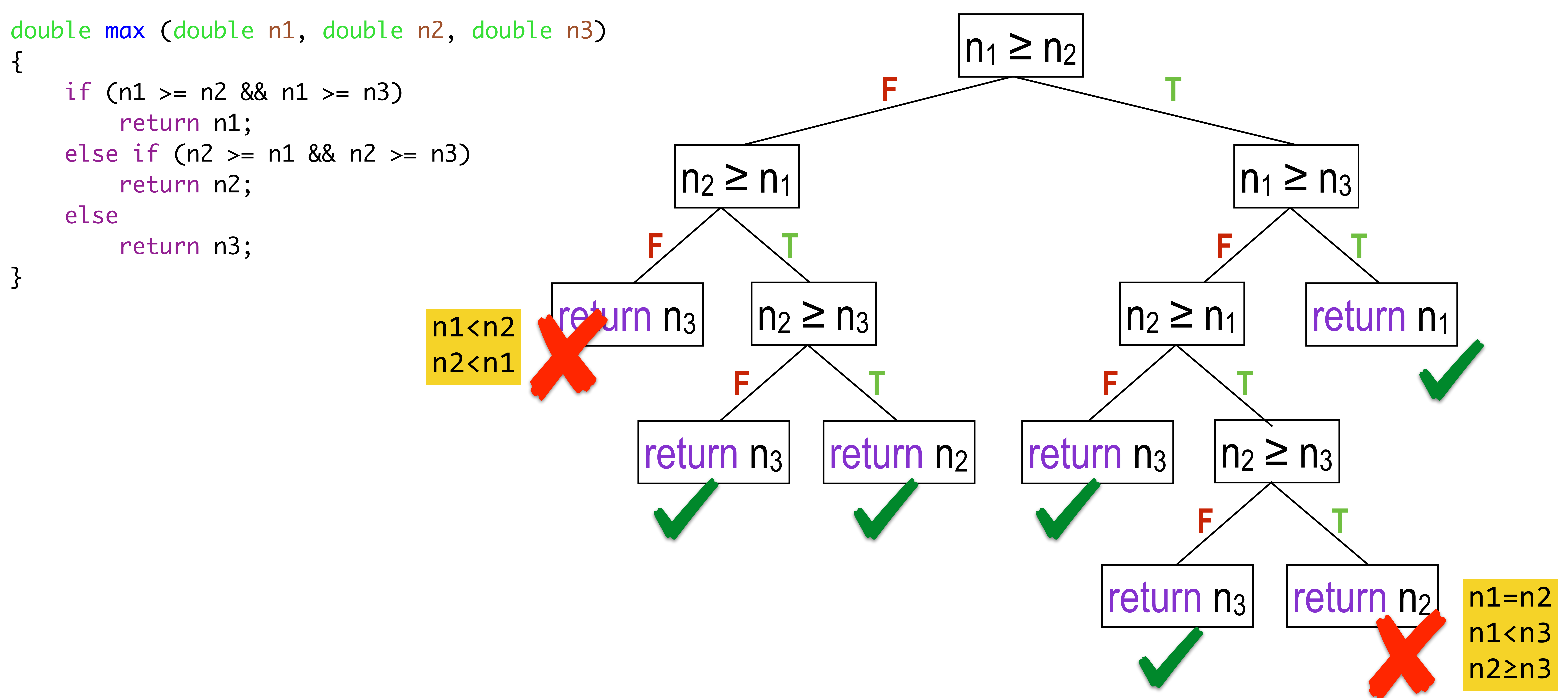
How many possible execution paths (behaviors) ?



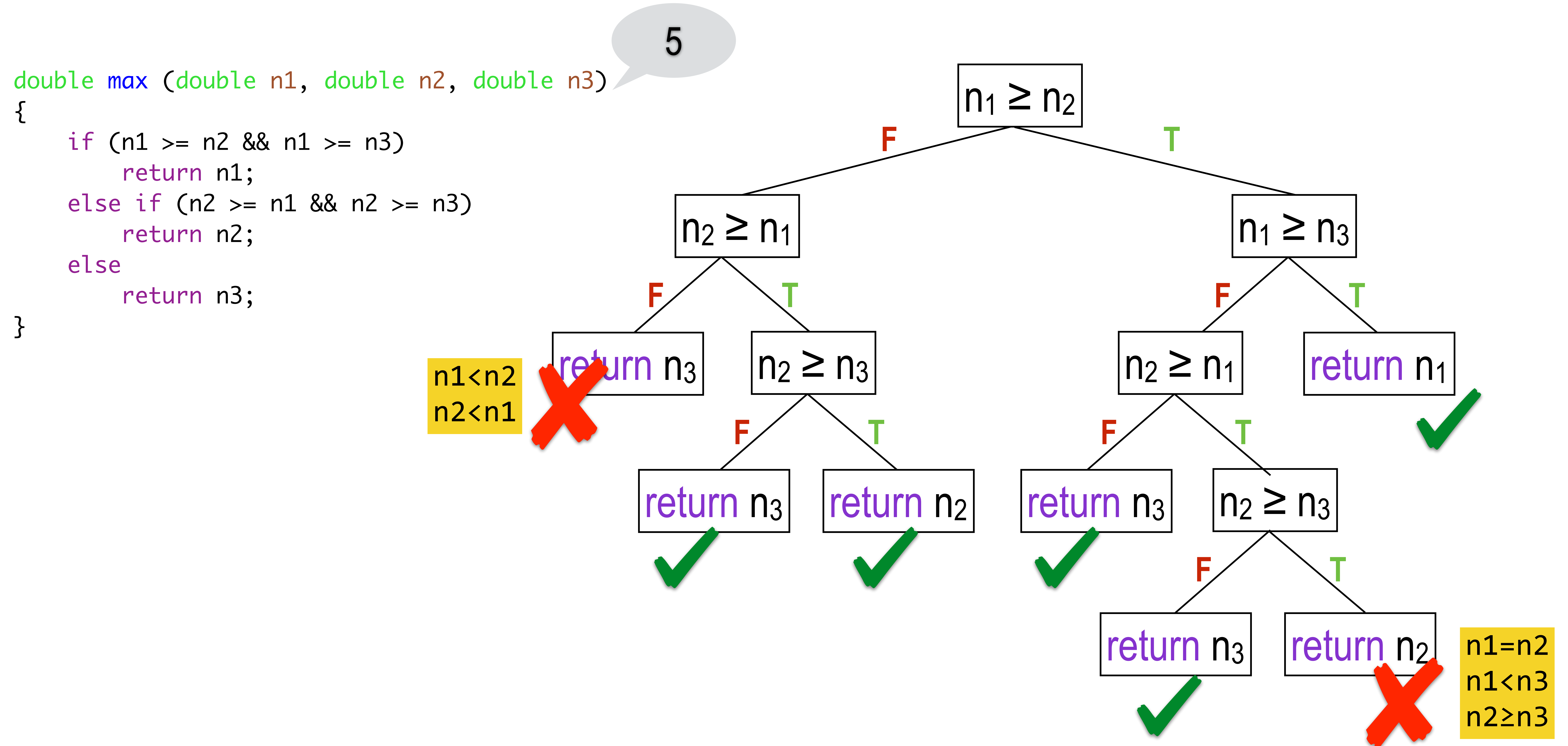
How many possible execution paths (behaviors) ?



How many possible execution paths (behaviors) ?



How many possible execution paths (behaviors) ?

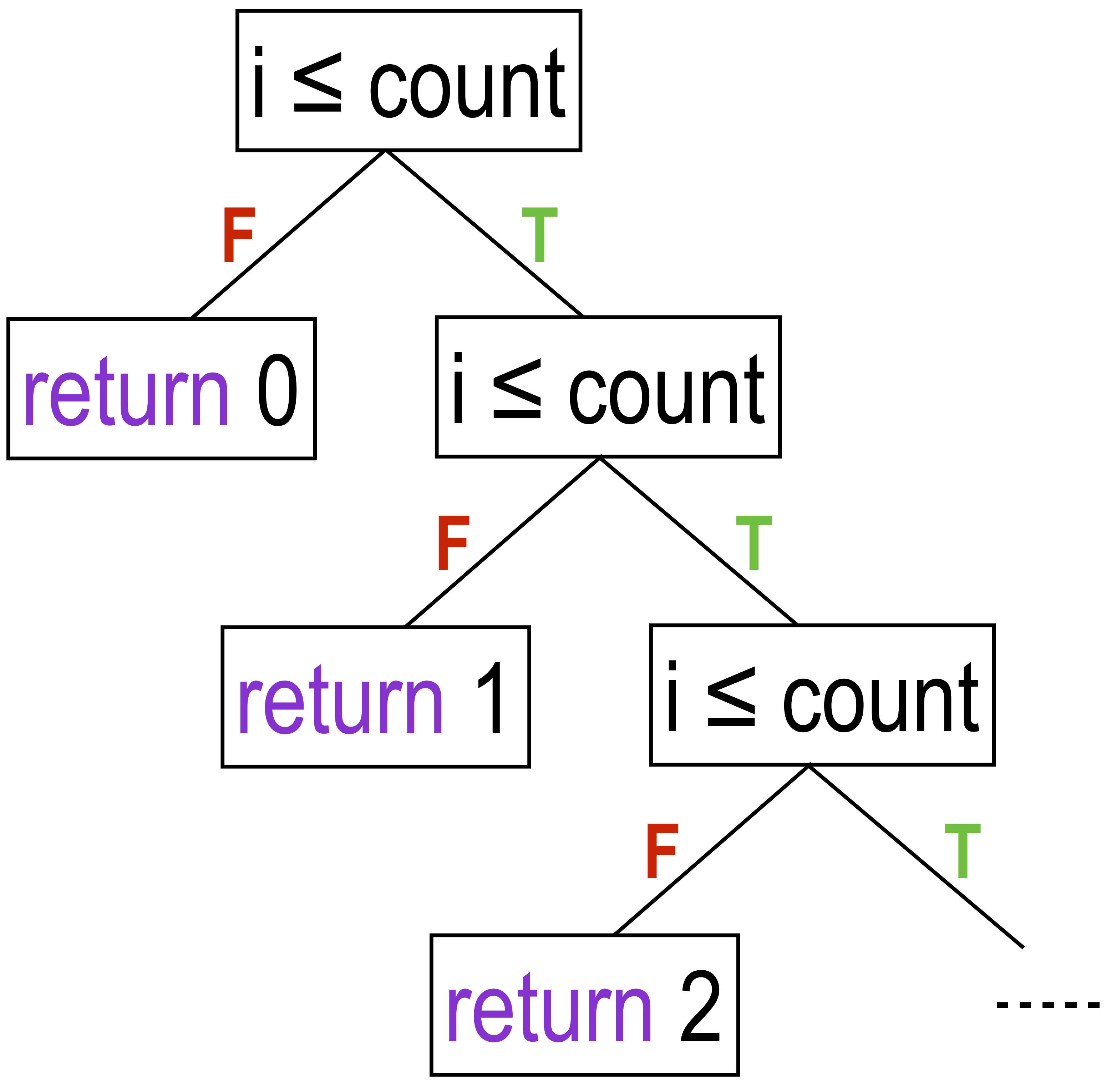


How many execution paths (possible behaviors) ?

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

How many execution paths (possible behaviors) ?

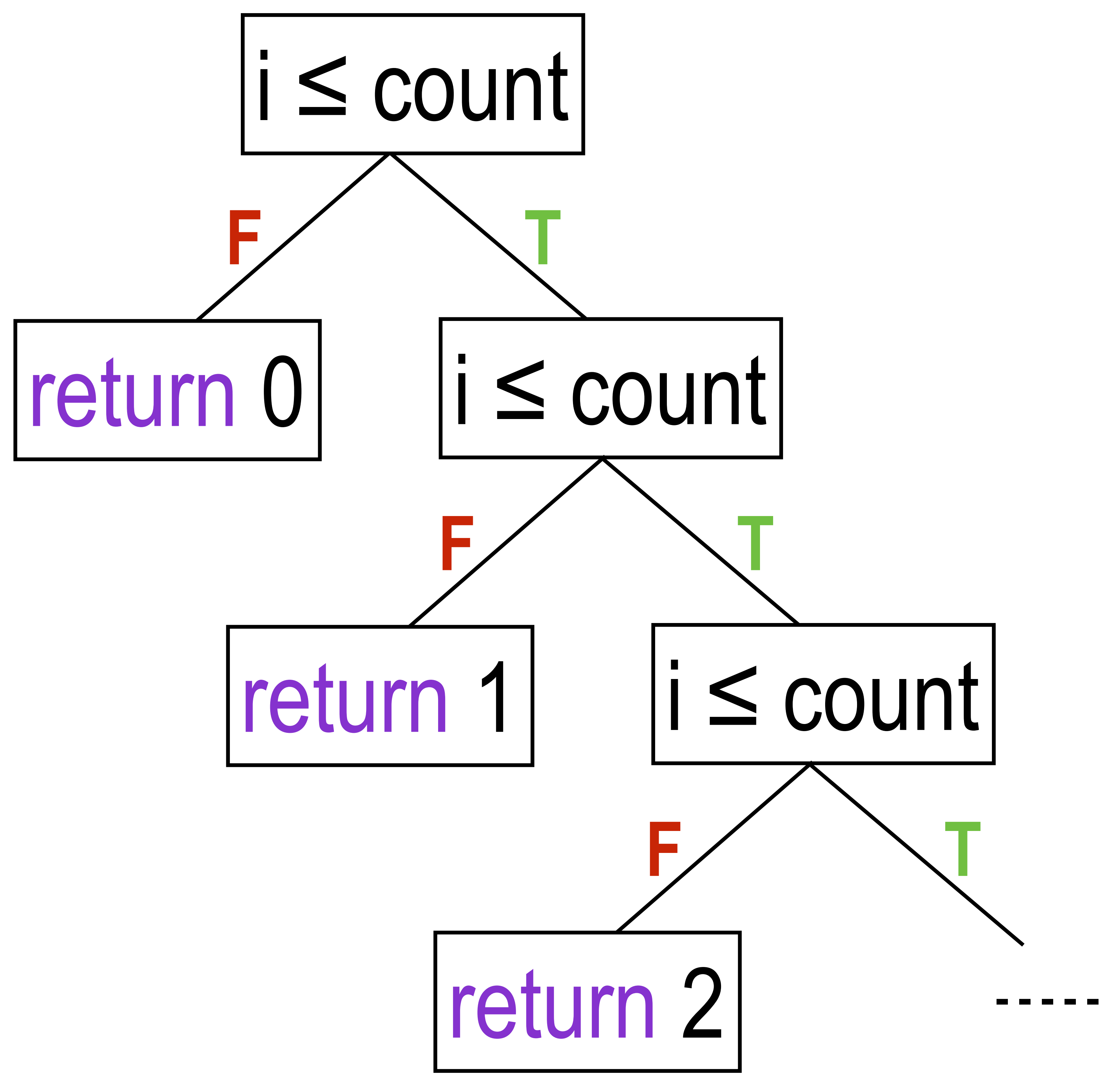
```
int iterationsByte (byte count)
{
    int iterations=0;
    for (int i = 1; i <= count; ++i) {
        ++iterations;
    }
    return iterations;
}
```



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;
}
```

How many execution paths (possible behaviors) ?

32768

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

How many execution paths (possible behaviors) ?

32768

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

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

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

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

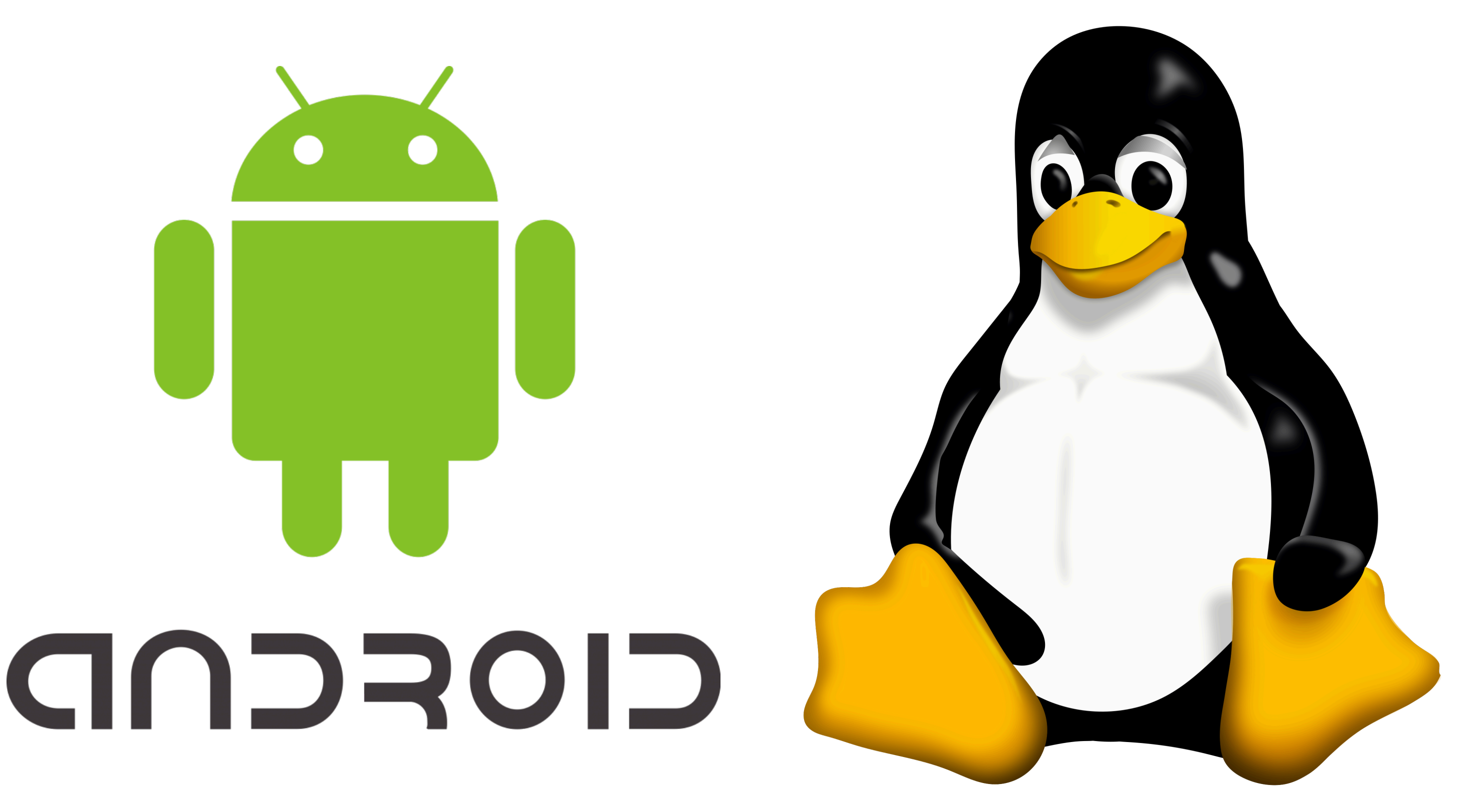
- entry-level electric vehicle (Chevy Volt) ~10 MLOC

- Android operating system ~a few tens of MLOC

- the Large Hadron Collider ~50 MLOC
- all car software in a high-end car ~100 MLOC

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

Software Testing



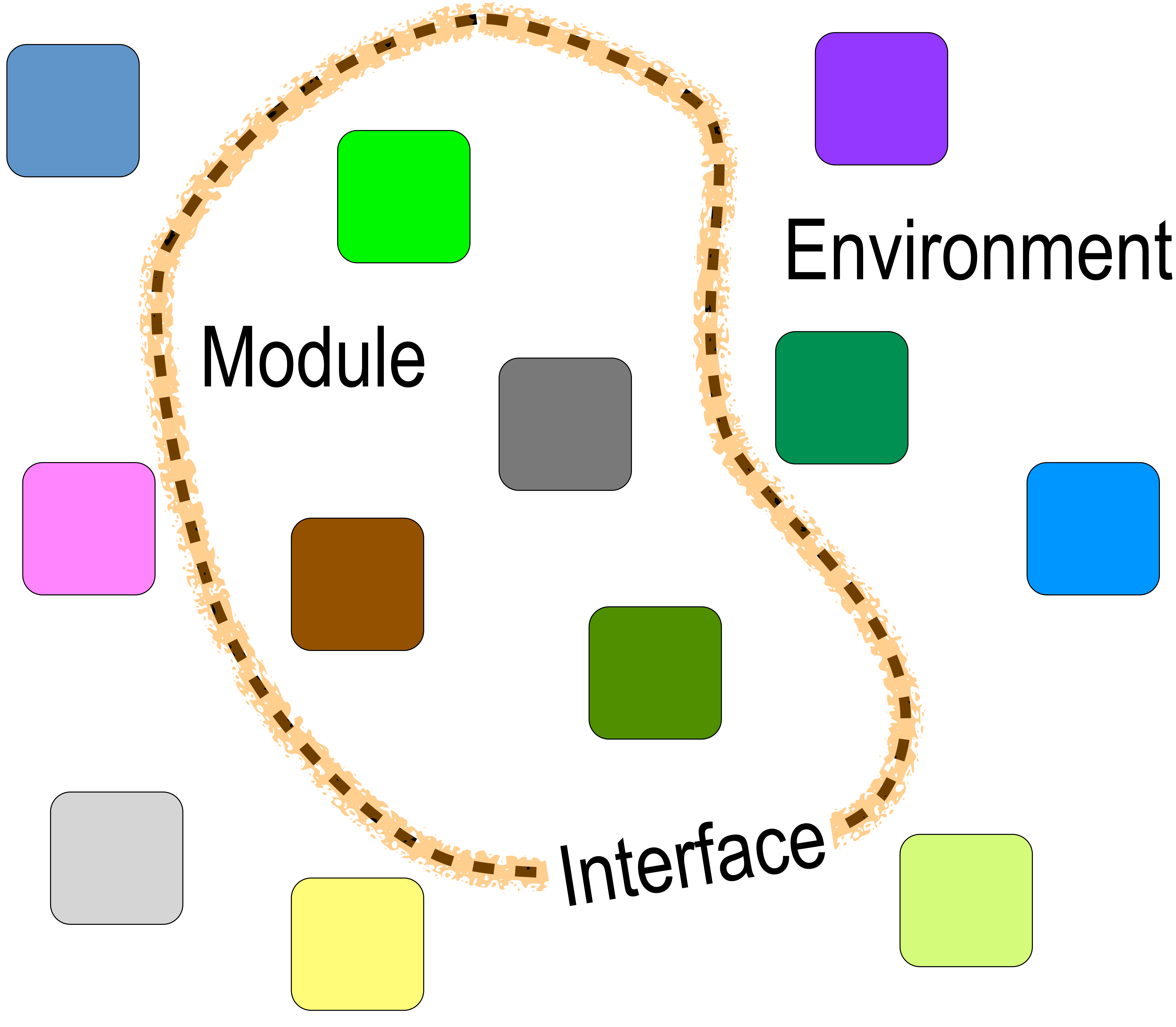
way more than 5,000,000 lines of code[1] (LOC) ⇒ > $2^{500,000}$ paths

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

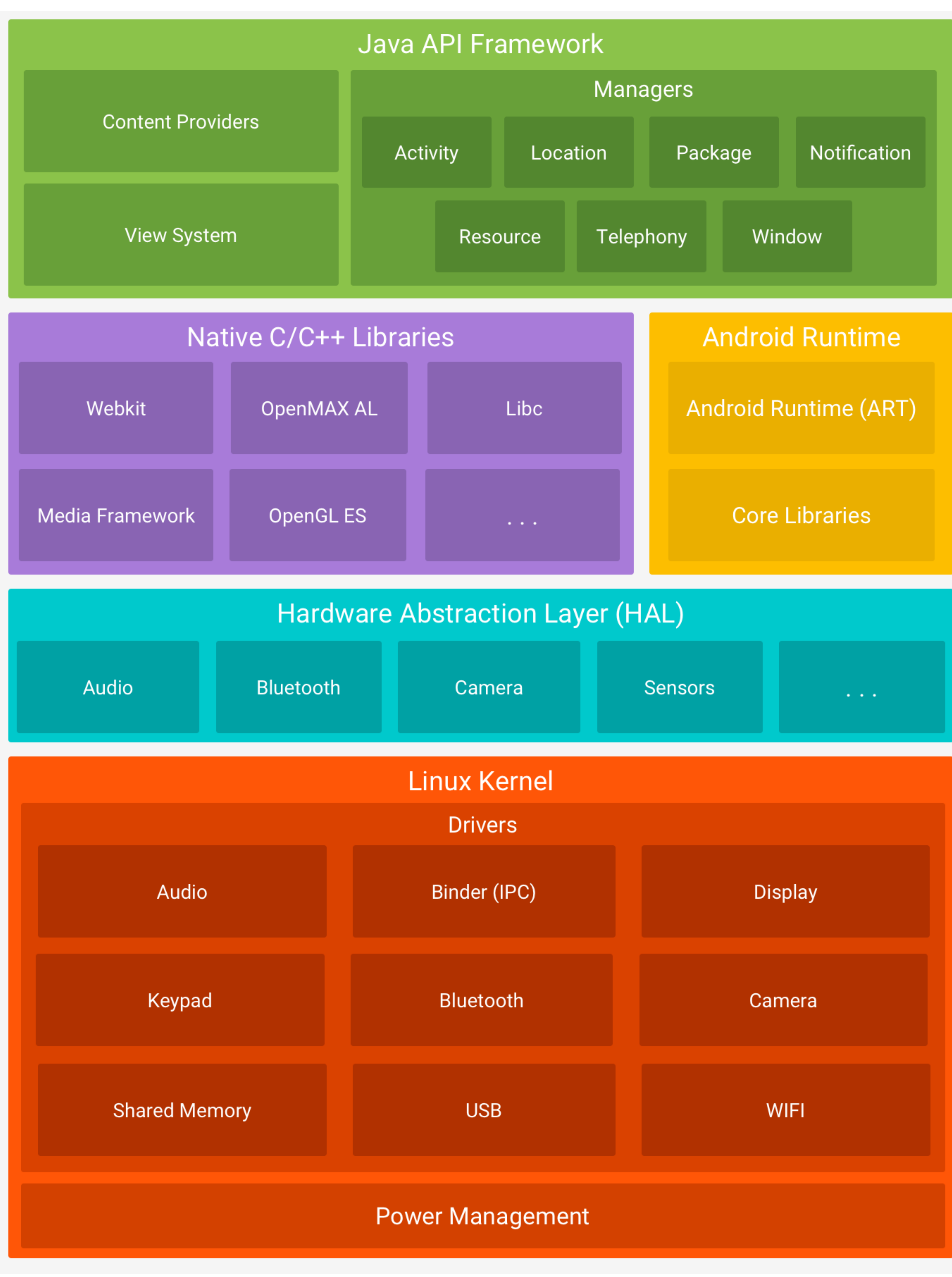
[1] https://www.visualcapitalist.com/millions-lines-of-code/

Testing Programs ≠ Testing Software

Modularity

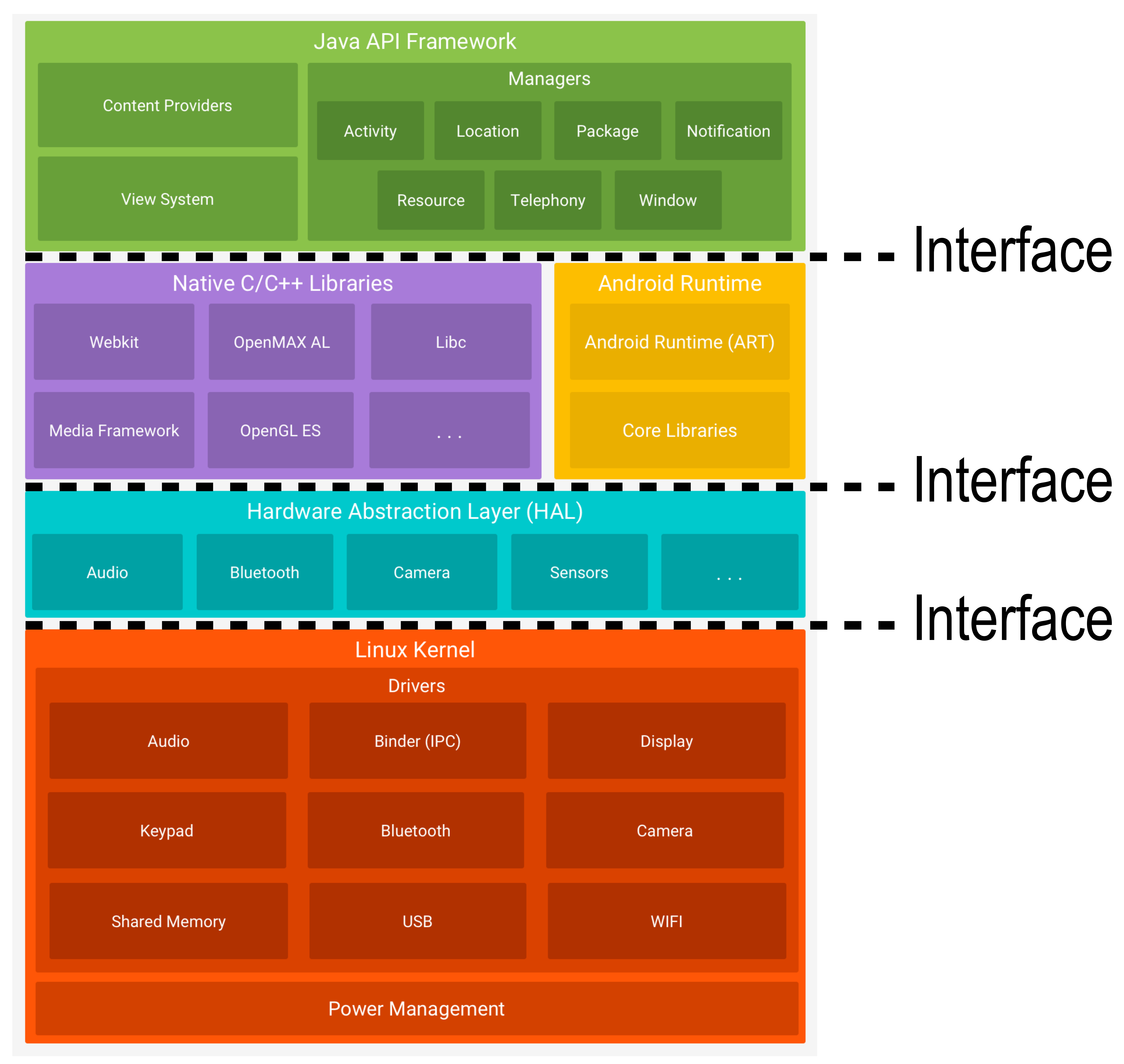


Layering



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

Layering



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

Conclusion

> 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)*

- 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?

Outline

- Recap of Testing
- Cost of Bugs
- How well can we test?
- Coverage Metrics
- Test-Driven Development (TDD) — *online*
- Behavior-Driven Development (BDD) — *online*

Measuring Test Quality

```
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
    }
}
```

```
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
    }
}
```

```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
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
    }
}
```

```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
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
    }
}
```

```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
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 = 7 / 8 = 87.5%

```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 = 7 / 8 = 87.5%

```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 = 7 / 8 = 87.5%

```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
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
    }
}
```

```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
    }
}
```

```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
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
    }
}
```

```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
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
    }
}
```

```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
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 = 2 / 4 = 50%

```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 > threshold) {
☐       return discountedTotal
    } else {
✅      return threshold
    }
}
```

branch coverage = 2 / 4 = 50%

```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 > threshold) {
✅ ☐      return discountedTotal
    } else {
✅        return threshold
    }
}
```

branch coverage = 2 / 4 = 50%

```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 > threshold) {
✅      return discountedTotal
    } else {
✅      return threshold
    }
}
```

branch coverage = 3 / 4 = 75%

```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 = 3 / 4 = 75%

```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 = 3 / 4 = 75%

```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%

```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)
}
```

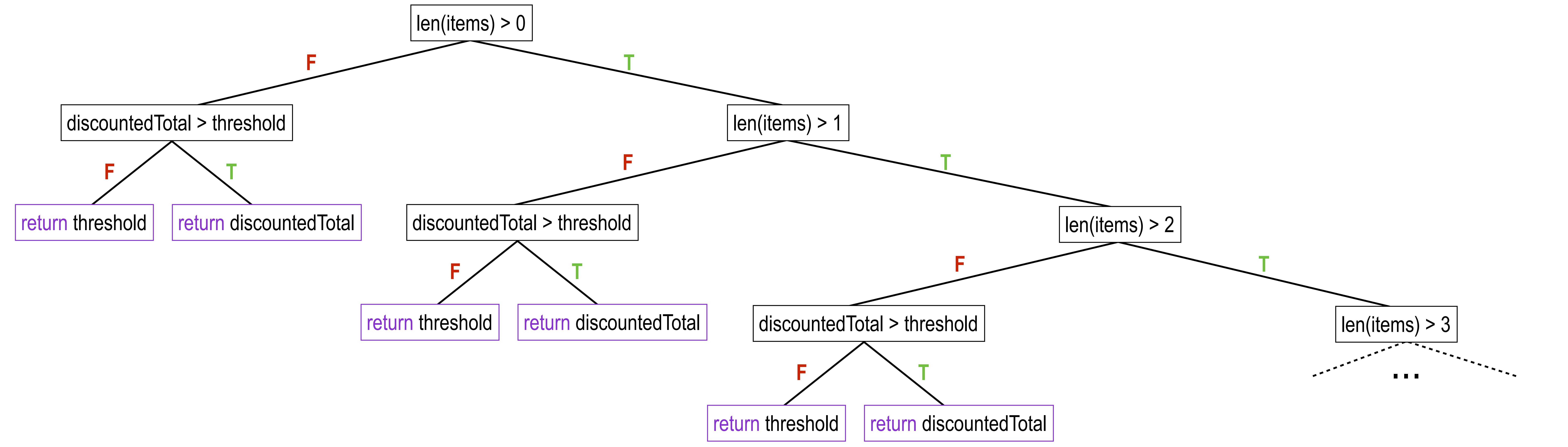
```kotlin
fun totalAfterDiscounts(items: List<Item>, threshold: Double,
                        weeklyDiscounts: Map<String, Double>): Double {
    var discountedTotal = 0.0
    for (item in items) {
✅      val discountFactor = weeklyDiscounts.getOrDefault(item.name, 1.0)
        val itemPrice = item.price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

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

branch coverage = 4 / 4 = 100%

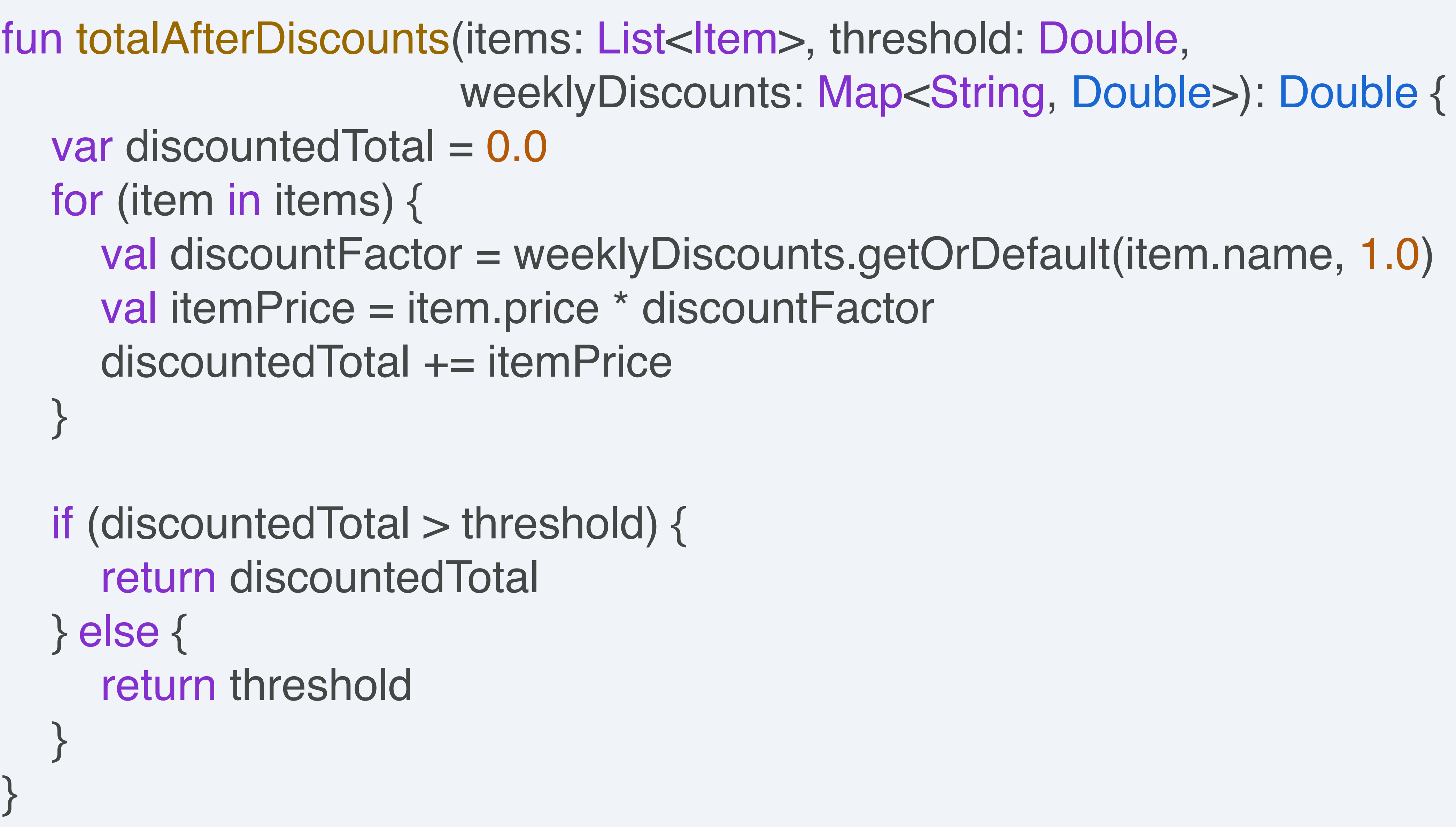
Map<K,V>.getOrDefault() implies a branch: *item.name* is found or not found. Here, we abstract that concern away.

```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
    }
}
```

```
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
    }
}
```



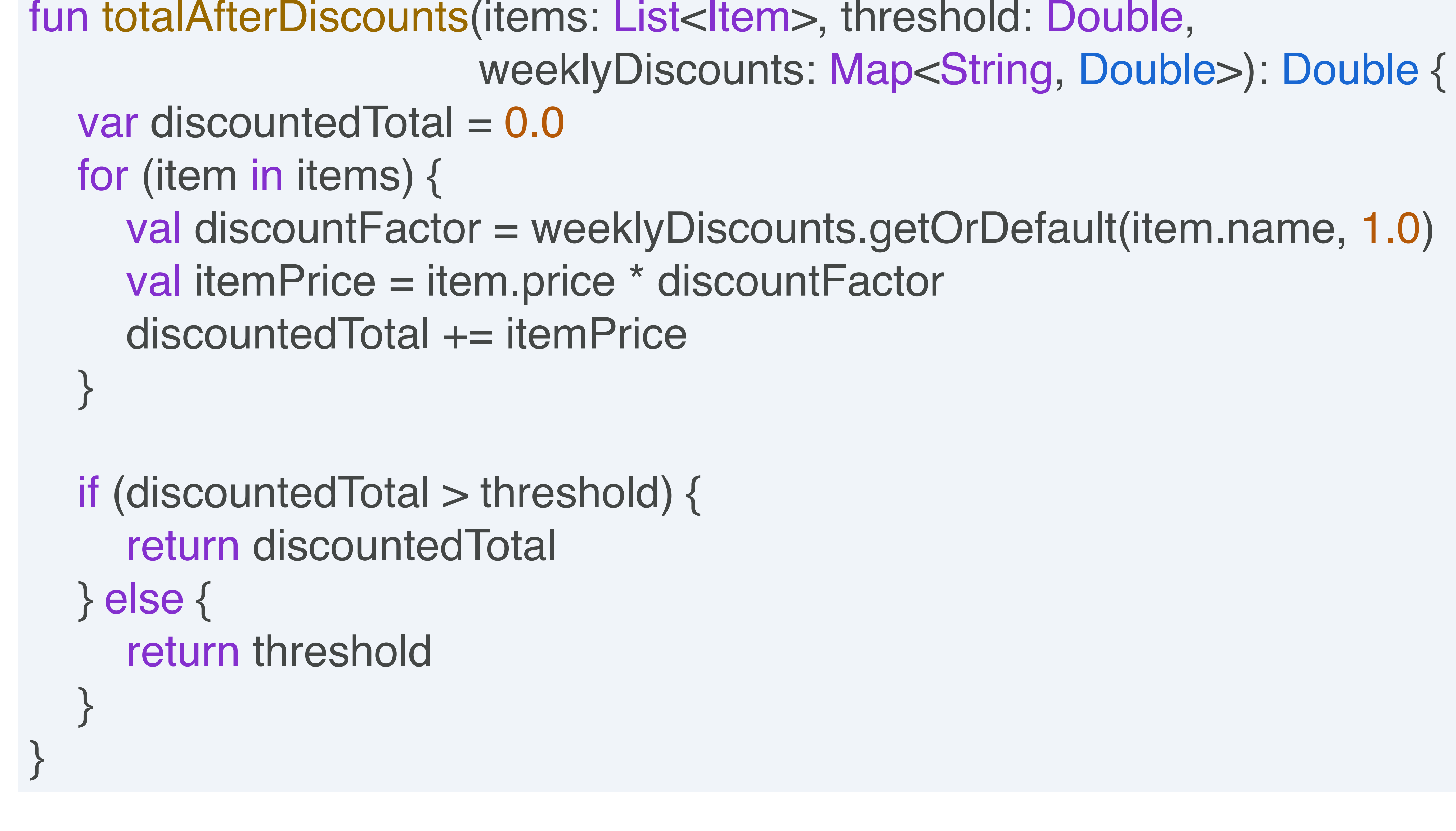
```kotlin
@Test
fun testApplyThreshold() {
    val items = listOf(Item("Soup", 5.0))
    val threshold = 10.0
    val weeklyDiscounts = emptyMap<String, Double>() // no discounts
    val result = totalAfterDiscounts(items, threshold, weeklyDiscounts)

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



```kotlin
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
    }
}
```

- len(items) > 0
  - F: discountedTotal > threshold
    - F: return threshold
    - T: return discountedTotal
  - T: len(items) > 1
    - F: discountedTotal > threshold
      - F: return threshold
      - T: return discountedTotal
    - T: len(items) > 2
      - F: discountedTotal > threshold
        - F: return threshold
        - T: return discountedTotal
      - T: len(items) > 3
        - ...

```
@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
}
```

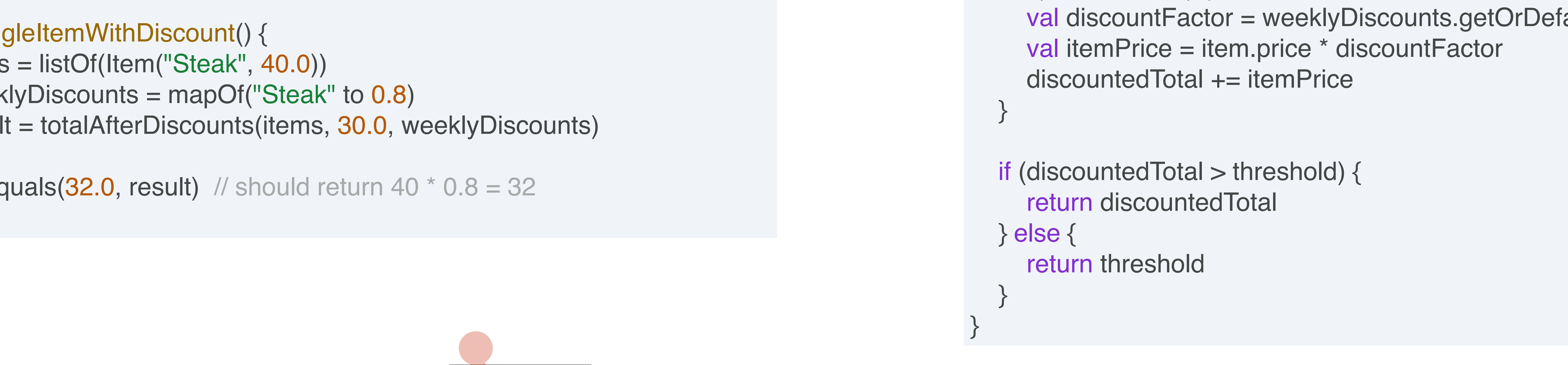


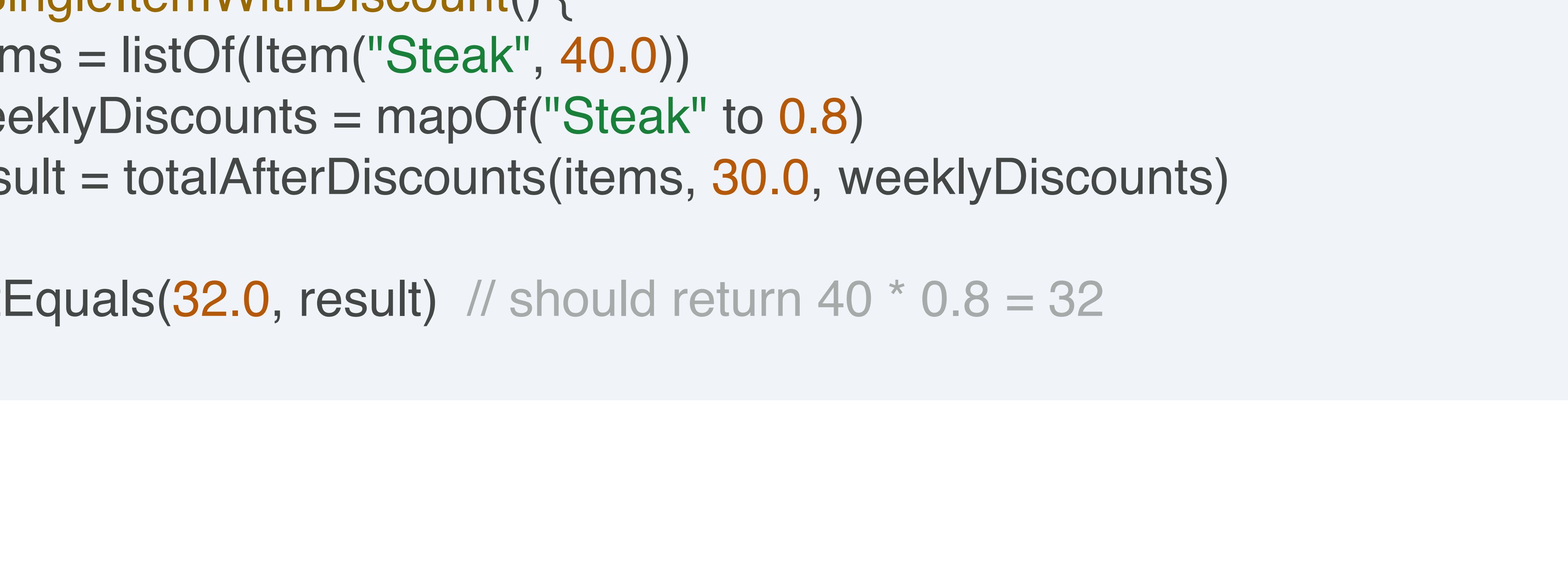
```
fun totalAfterDiscounts(items: List<Item>, threshold: Double,
                        weeklyDiscounts: Map<String, Double>): Double {
    var discountedTotal = 0.0
    for (item in items) {
        val discountFactor = weeklyDiscounts.getOrDefault(item.name, 1.0)
        val itemPrice = item.price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

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

len(items) > 0
F
T
discountedTotal > threshold
F
T
return threshold
return discountedTotal
len(items) > 1
F
T
discountedTotal > threshold
F
T
return threshold
return discountedTotal
len(items) > 2
F
T
discountedTotal > threshold
F
T
return threshold
return discountedTotal
len(items) > 3
...

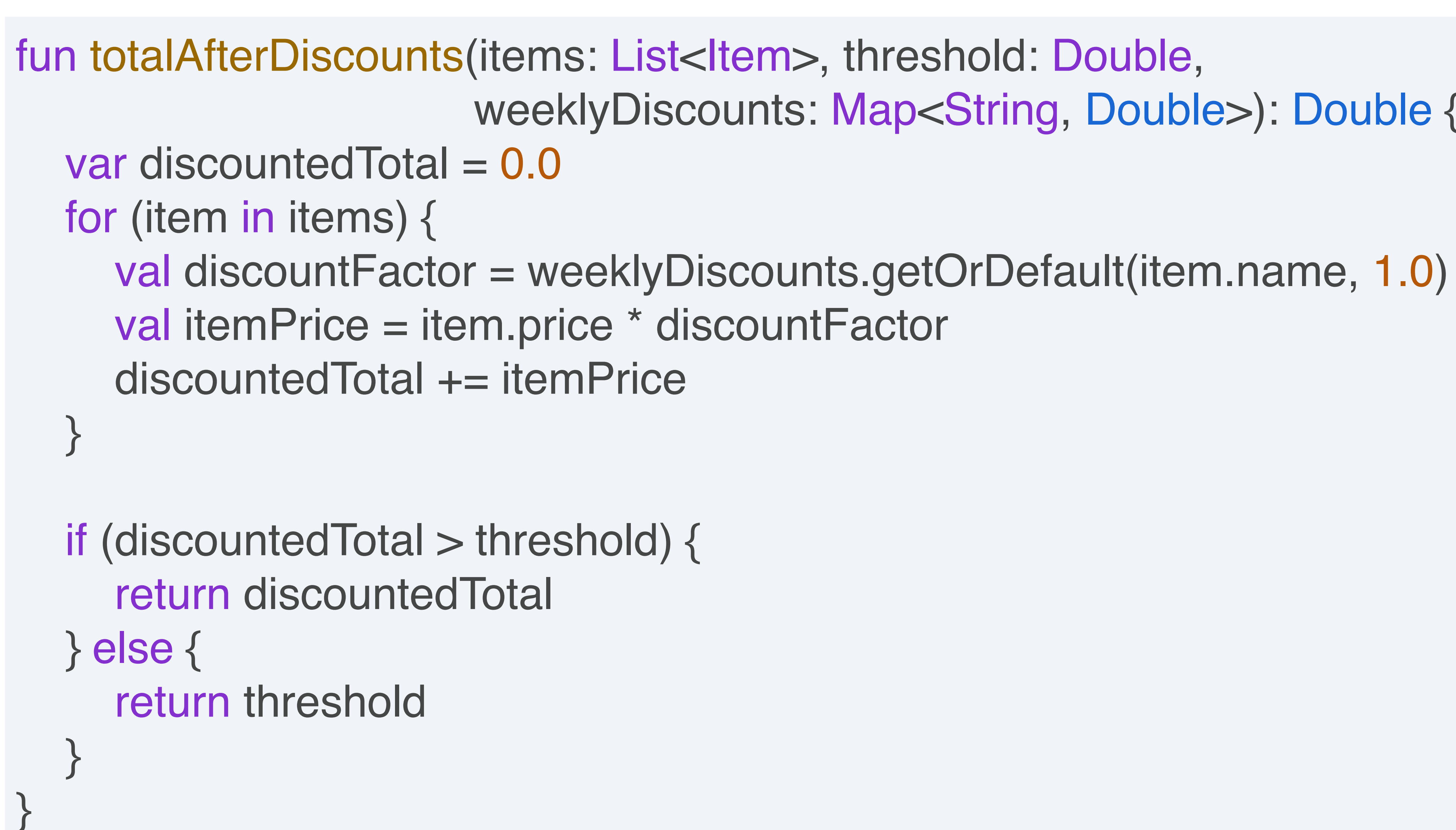
```kotlin
@Test
fun testSingleItemWithDiscount() {
    val items = listOf(Item("Steak", 40.0))
    val weeklyDiscounts = mapOf("Steak" to 0.8)
    val result = totalAfterDiscounts(items, 30.0, weeklyDiscounts)

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





```
@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
}
```

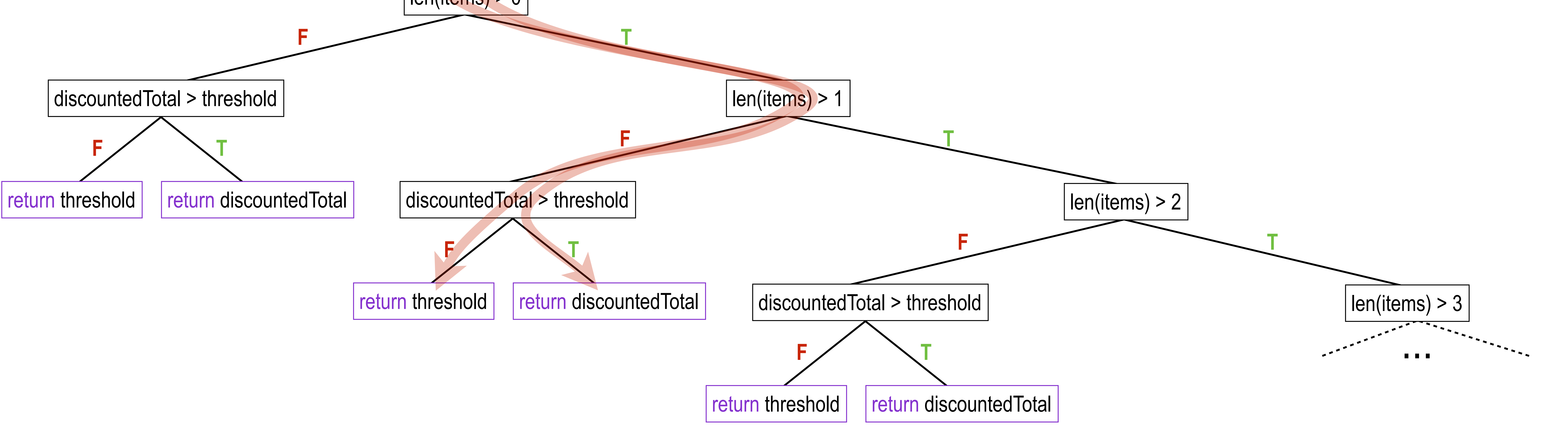


```
fun totalAfterDiscounts(items: List<Item>, threshold: Double,
                        weeklyDiscounts: Map<String, Double>): Double {
    var discountedTotal = 0.0
    for (item in items) {
        val discountFactor = weeklyDiscounts.getOrDefault(item.name, 1.0)
        val itemPrice = item.price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

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

len(items) > 0
F
T
discountedTotal > threshold
len(items) > 1
F
T
F
T
return threshold
return discountedTotal
discountedTotal > threshold
len(items) > 2
F
T
F
T
return threshold
return discountedTotal
discountedTotal > threshold
len(items) > 3
F
T
...
return threshold
return discountedTotal

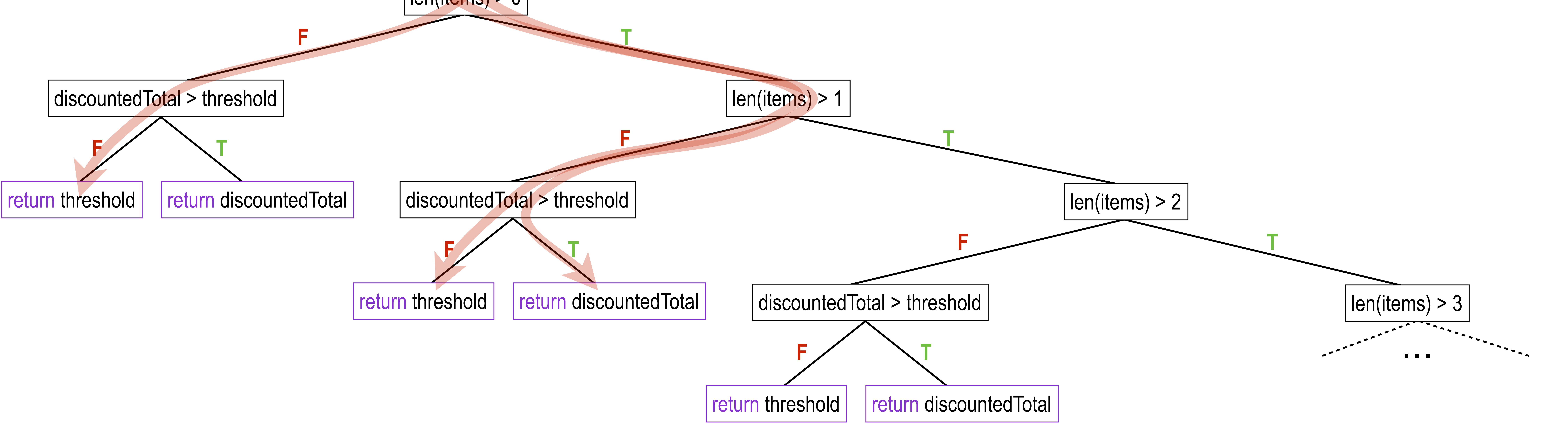
```kotlin
@Test
fun testNoItems() {
    val items = listOf<Item>()
    val threshold = 50.0
    val weeklyDiscounts = emptyMap<String, Double>()
    val result = totalAfterDiscounts(items, threshold, weeklyDiscounts)

    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
    }
}
```

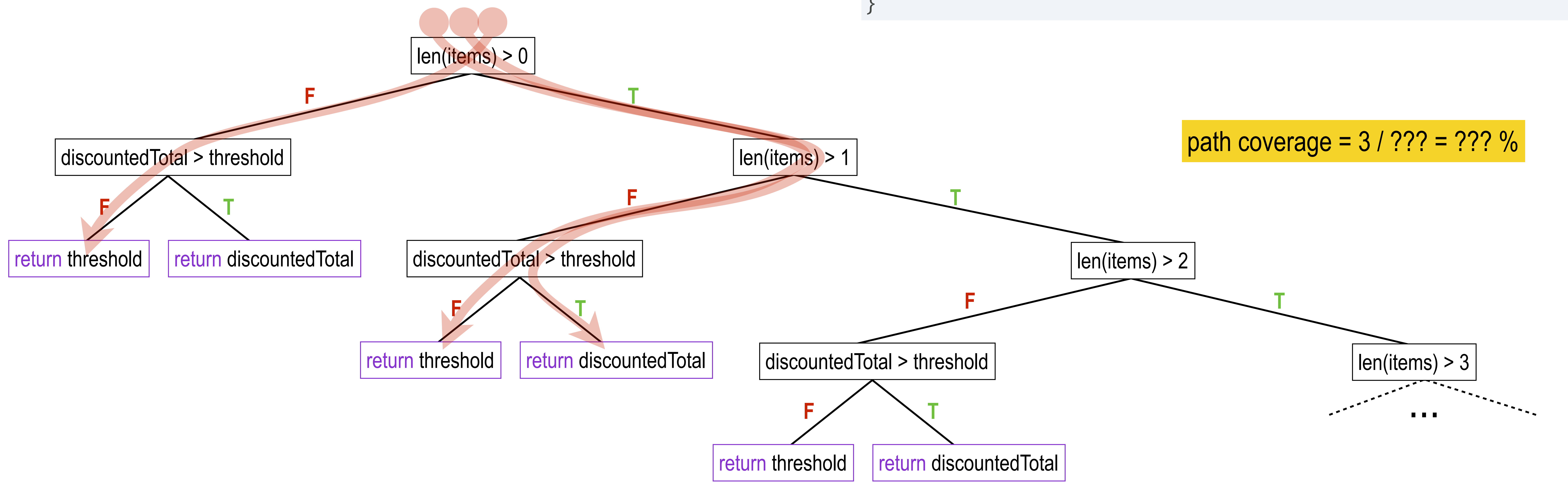


```kotlin
@Test
fun testNoItems() {
    val items = listOf<Item>()
    val threshold = 50.0
    val weeklyDiscounts = emptyMap<String, Double>()
    val result = totalAfterDiscounts(items, threshold, weeklyDiscounts)

    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
    }
}
```



```
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
    }
}
```



```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
    }
}
```

```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
    }
}
```


E-N+2P linearly independent paths

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
    }
}
```

Enter

discountedTotal=0.0

for (item in items)

discountFactor = weeklyDiscounts.getOrDefault(item.name, 1.0)
itemPrice = item.price * discountFactor
discountedTotal += itemPrice

discountedTotal > threshold)

return=discountedTotal

return=threshold

Exit

1

+1

```
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
    }
}
```

Enter

discountedTotal=0.0

for (item in items)

discountFactor = weeklyDiscounts.getOrDefault(item.name, 1.0)
itemPrice = item.price * discountFactor
discountedTotal += itemPrice

discountedTotal > threshold)

return=discountedTotal

return=threshold

Exit

1

+1

+1

```
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

Enter

discountedTotal=0.0

for (item in items)

discountFactor = weeklyDiscounts.getOrDefault(item.name, 1.0)
itemPrice = item.price * discountFactor
discountedTotal += itemPrice

discountedTotal > threshold)

return=discountedTotal

return=threshold

Exit

```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
1    fun totalAfterDiscounts(items: List<Item>, threshold: Double,
                                 weeklyDiscounts: Map<String, Double>): Double {
         var discountedTotal = 0.0
+1       for (item in items) {
             val discountFactor = weeklyDiscounts.getOrDefault(item.name, 1.0)
             val itemPrice = item.price * discountFactor
             discountedTotal += itemPrice
         }

+1       if (discountedTotal > threshold) {
             return discountedTotal
         } else {
             return threshold
         }
     }
```

cyclomatic complexity = 3

```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
1   fun totalAfterDiscounts(items: List<Item>, threshold: Double,
                            weeklyDiscounts: Map<String, Double>): Double {
        var discountedTotal = 0.0
+1      for (item in items) {
            val discountFactor = weeklyDiscounts.getOrDefault(item.name, 1.0)
            val itemPrice = item.price * discountFactor
            discountedTotal += itemPrice
        }

+1      if (discountedTotal > threshold) {
            return discountedTotal
        } else {
            return threshold
        }
    }
```

cyclomatic complexity = 3

```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
1    fun totalAfterDiscounts(items: List<Item>, threshold: Double,
                             weeklyDiscounts: Map<String, Double>): Double {
         var discountedTotal = 0.0
+1       for (item in items) {
             val discountFactor = weeklyDiscounts.getOrDefault(item.name, 1.0)
             val itemPrice = item.price * discountFactor
             discountedTotal += itemPrice
         }

+1       if (discountedTotal > threshold) {
             return discountedTotal
         } else {
             return threshold
         }
     }
```

cyclomatic complexity = 3

```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
1   fun totalAfterDiscounts(items: List<Item>, threshold: Double,
                            weeklyDiscounts: Map<String, Double>): Double {
        var discountedTotal = 0.0
+1      for (item in items) {
            val discountFactor = weeklyDiscounts.getOrDefault(item.name, 1.0)
            val itemPrice = item.price * discountFactor
            discountedTotal += itemPrice
        }

+1      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
- 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
- Test-Driven Development (TDD) — *online*
- Behavior-Driven Development (BDD) — *online*

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()
}
```

https://towardsdev.com/kotlin-code-coverage-with-jacoco-5f5fb922afd4

JaCoCo Instrumentation (conceptual)

```kotlin
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()
}
```

```kotlin
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)
}
```

https://towardsdev.com/kotlin-code-coverage-with-jacoco-5f5fb922afd4

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

https://towardsdev.com/kotlin-code-coverage-with-jacoco-5f5fb922afd4

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

https://towardsdev.com/kotlin-code-coverage-with-jacoco-5f5fb922afd4

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

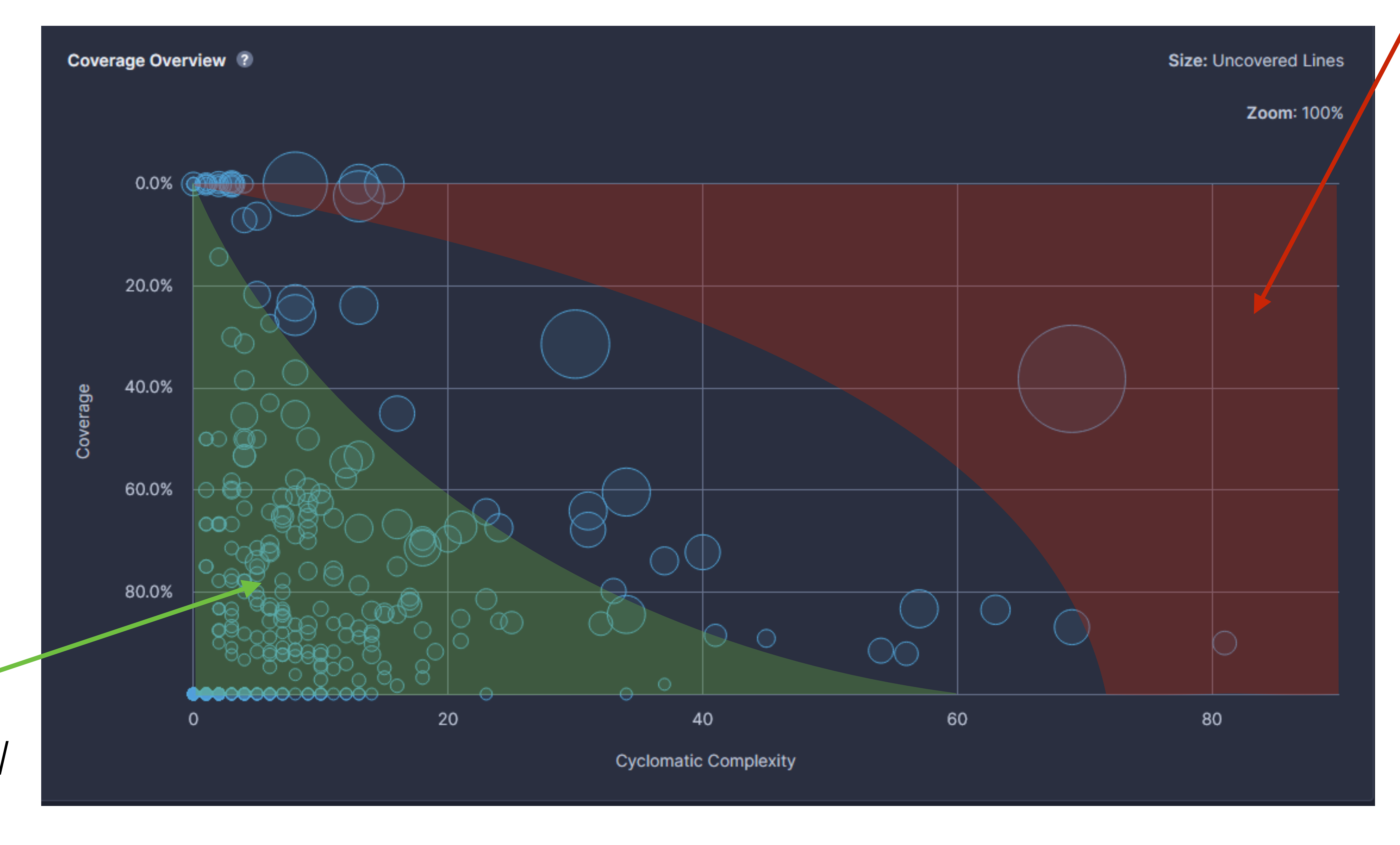
Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
MainKt		0%		n/a	1	1	4	4	1	1	1	1
JacocoDemo		57%		100%	1	4	2	4	1	3	0	1
Total	31 of 42	26%	0 of 2	100%	2	5	6	8	2	4	1	2

Main.kt

```kotlin
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.
```

https://towardsdev.com/kotlin-code-coverage-with-jacoco-5f5fb922afd4

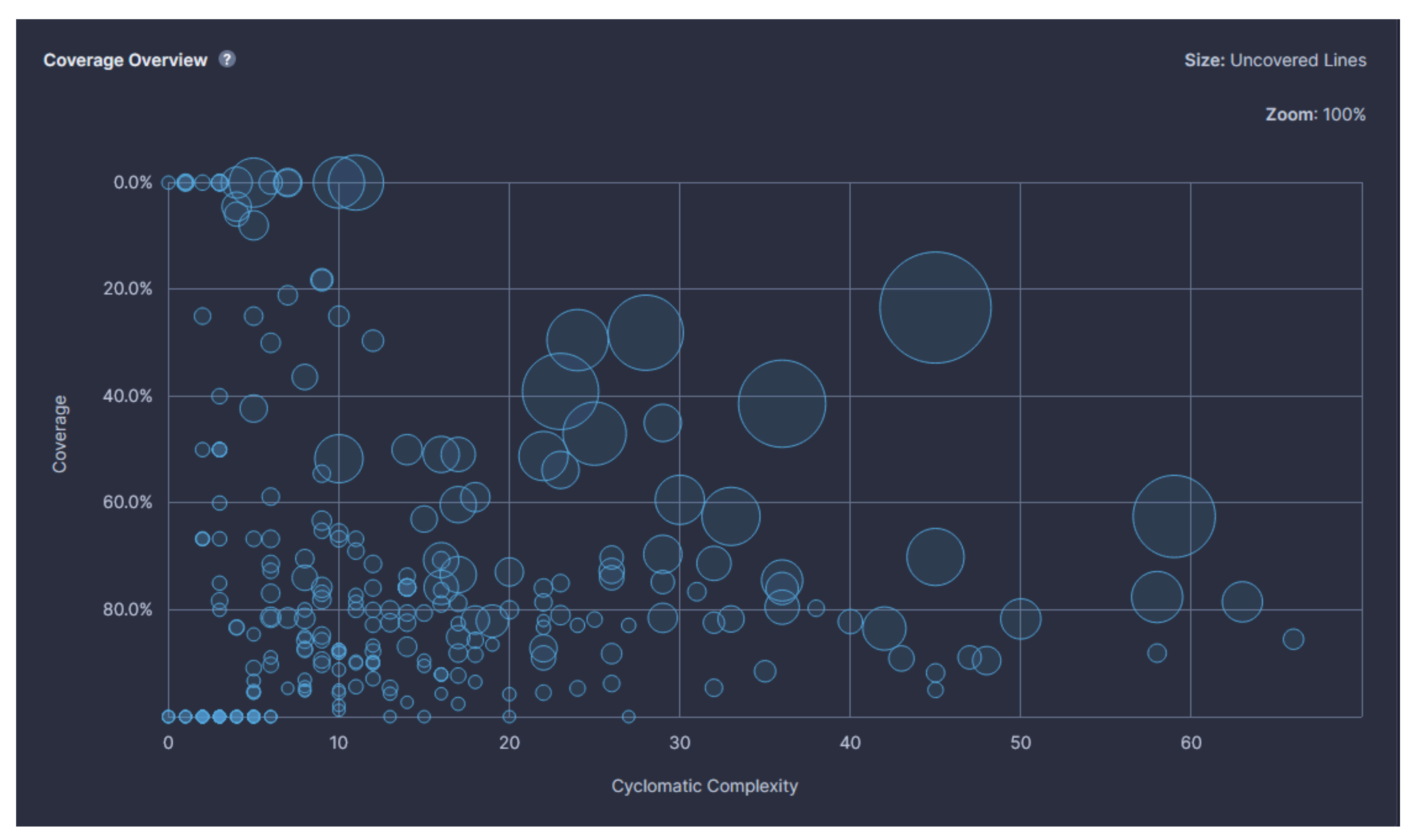
Tool Example: Sonar Cloud



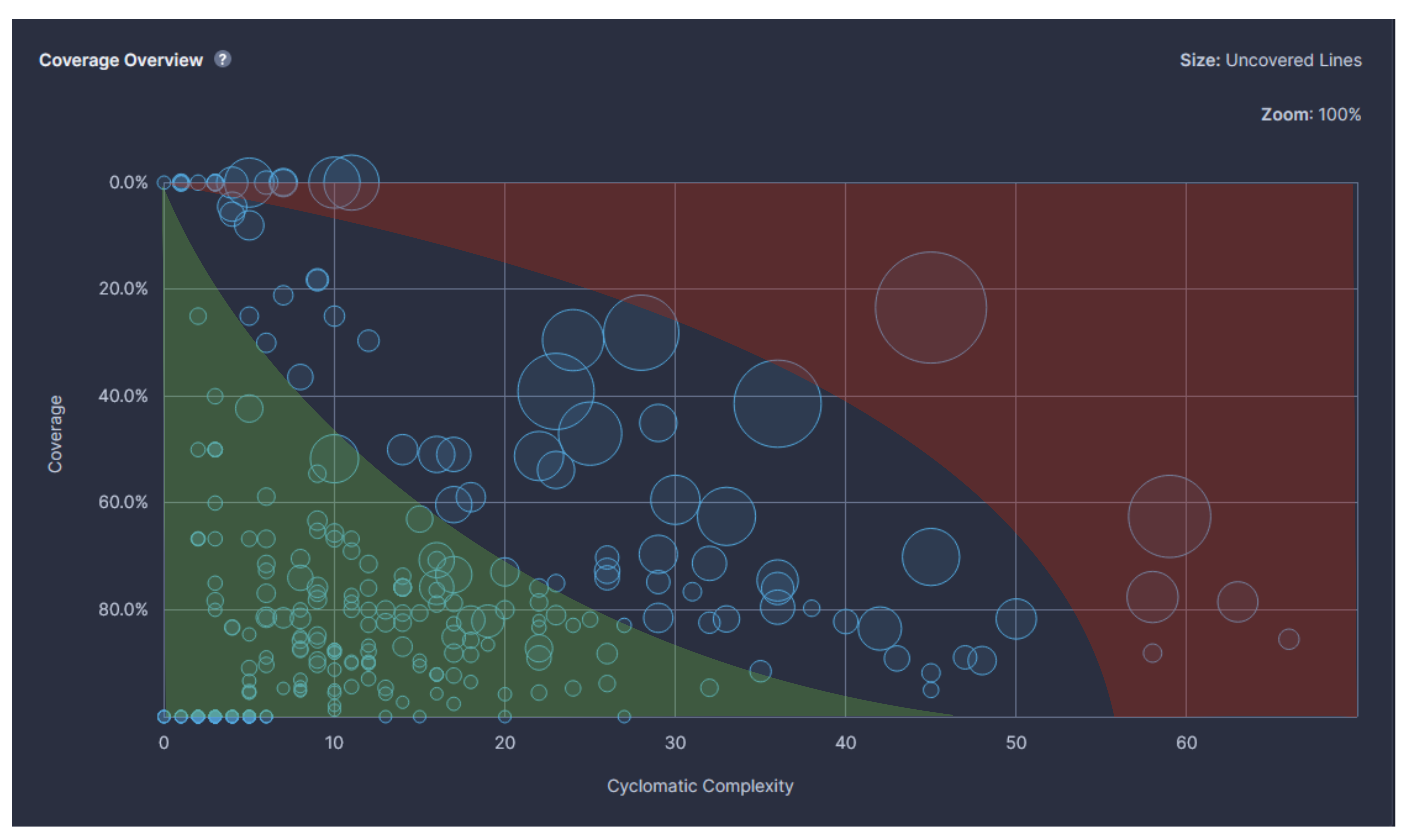
Bad:
High Complexity/
Low Coverage

Good:
Low Complexity/
High Coverage

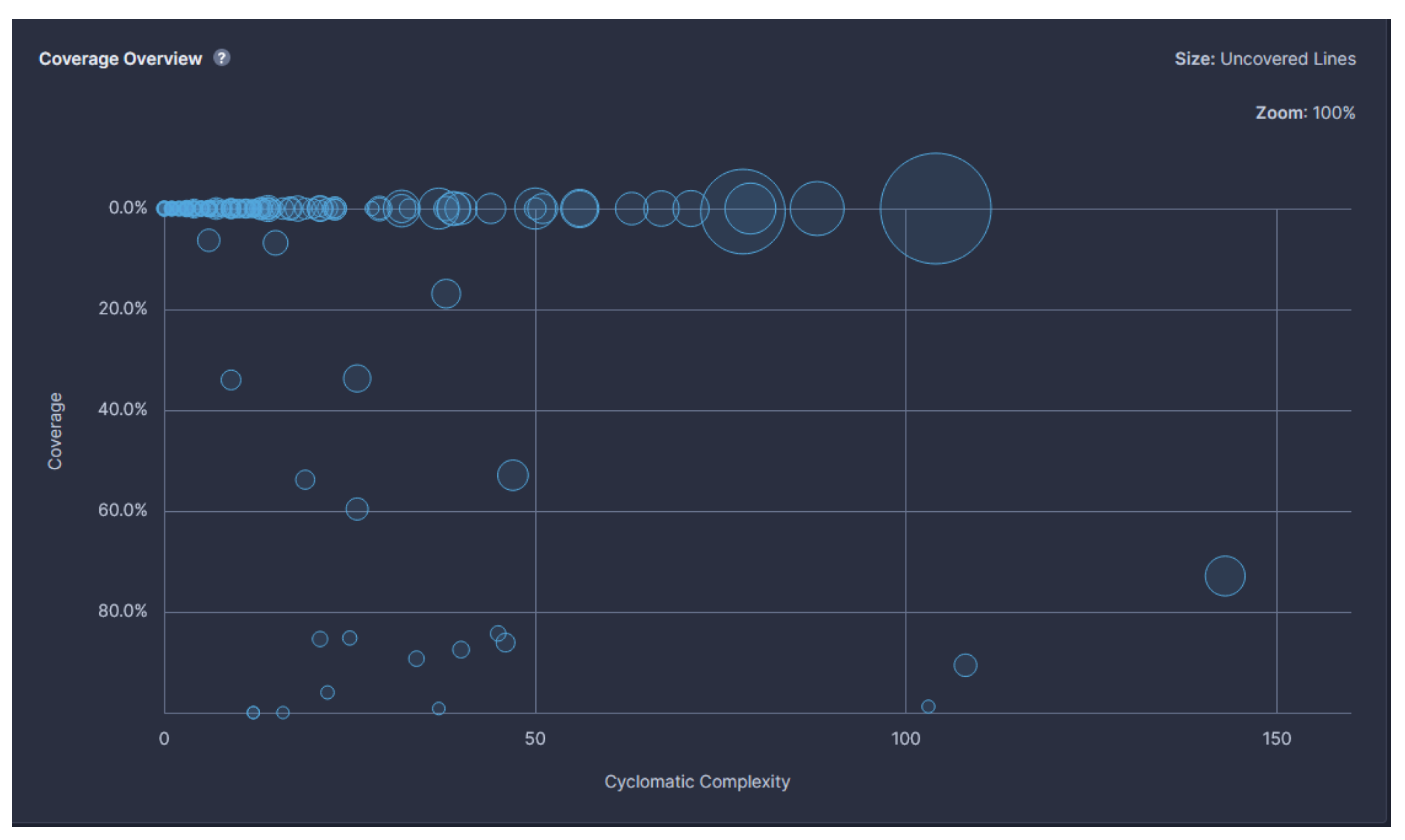
Coverage vs. Complexity



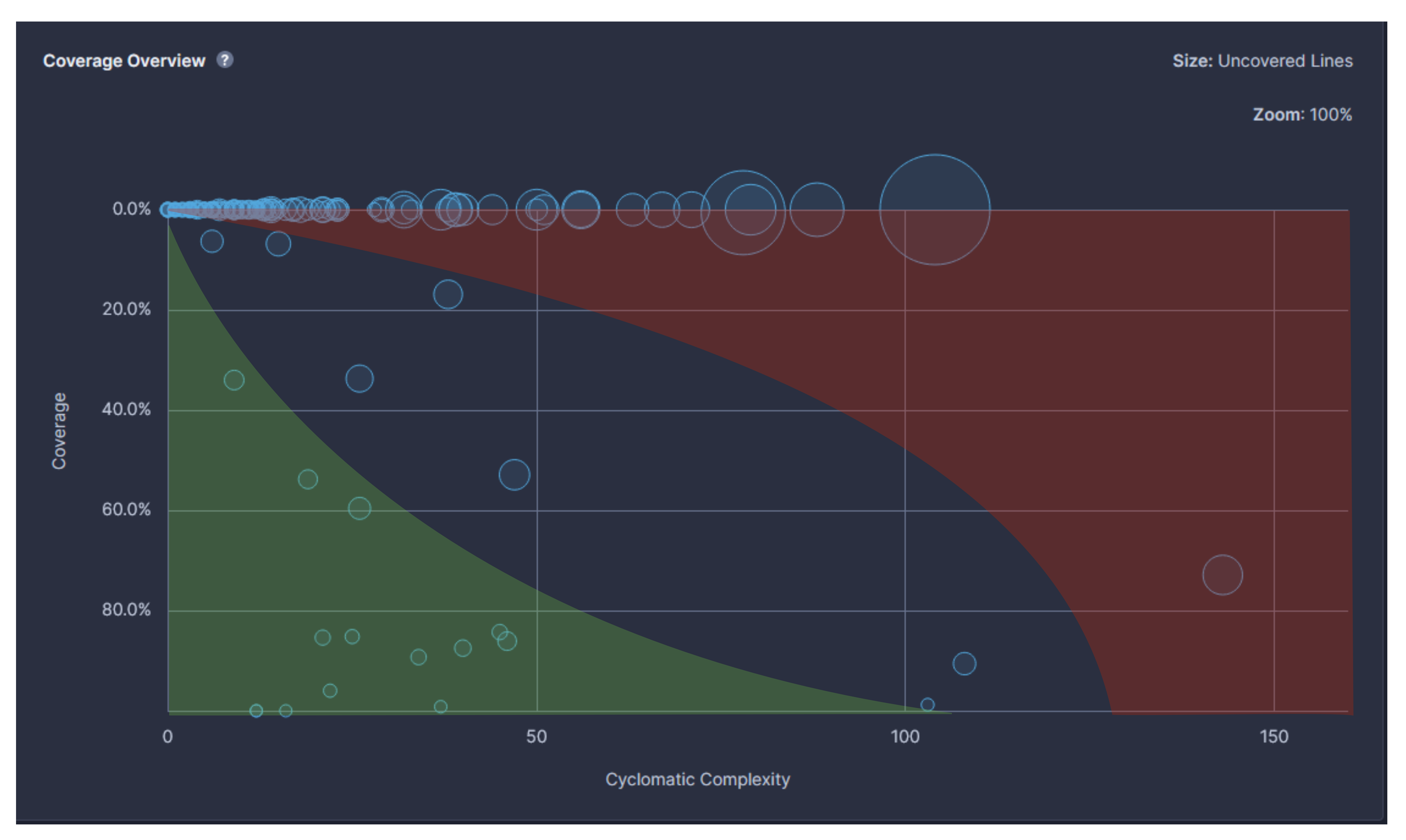
Coverage vs. Complexity



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

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 $\Rightarrow$ well-exercised codebase $\nRightarrow$ 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
- Dirty Pipe
- Log4Shell
- SolarWinds Orion Platform
- WannaCry Ransomware Attack
- Cloudbleed
- Heartbleed
- Knight Capital Group Trading Disaster
- DNS Cache Poisoning
- Y2K Bug (Year 2000)
- Mars Climate Orbiter Disintegration
- Ariane 5 Rocket Flight 501
- Patriot Missile
- Therac-25 Radiation Therapy Machine

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

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
- Test-Driven Development (TDD) — *online*
- Behavior-Driven Development (BDD) — *online*