# ANDROID TESTING

Filippo De Pretto - Wishew

# UN GRAZIE A GIUSEPPE

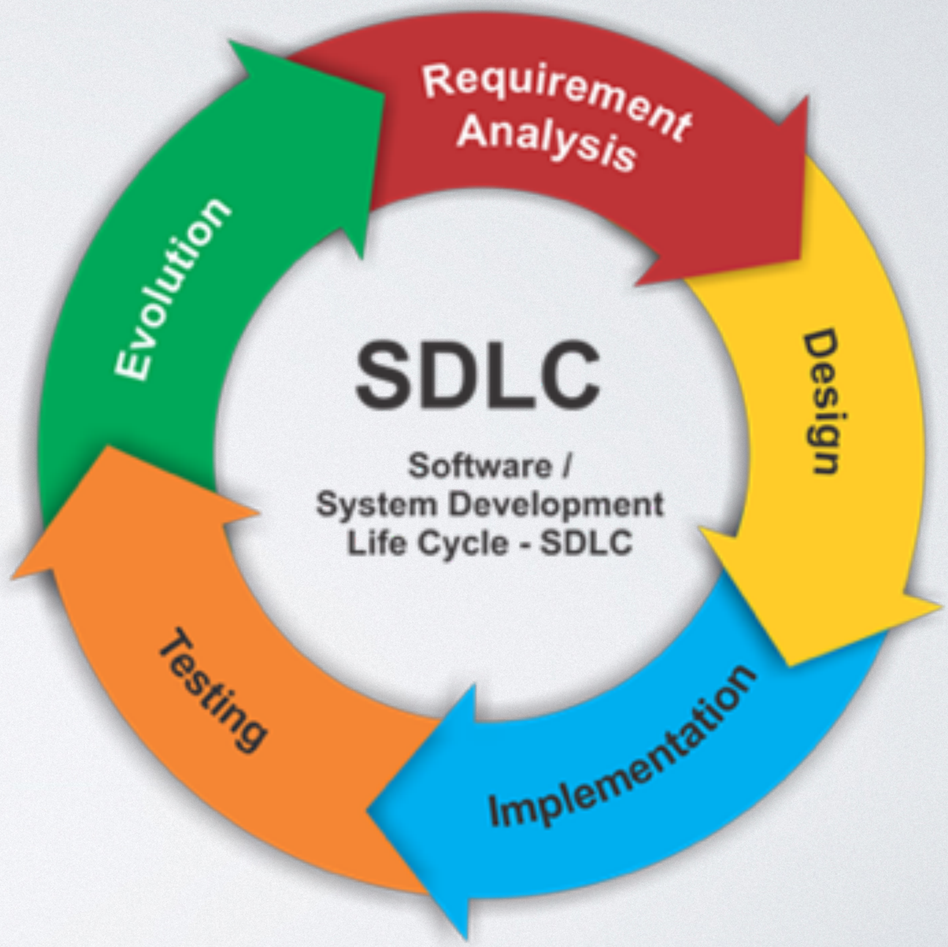
Grazie a **Giuseppe** abbiamo 10 partecipanti al workshop pratico :-)

# CHI SONO

- A Padova inizio a sviluppare in **Waterfall** con solo **test manuali**
- A giugno 2017 decido di trasferirmi per imparare a fare i test con **XPeppers** e lo sviluppo **Agile.**
- Da maggio 2018 inizio a usare i test su **Android**

# CICLO DI SVILUPPO DEL SOFTWARE

- Il **testing** è una fase importante quanto lo **sviluppo** del software.
- E' una competenza che spetta anche agli **sviluppatori** e non solo ai **tester** che spesso validano il software come black-box
- Nell'**Agile** con la **Continuous Integration** e la **Continuous Delivery** l'automated testing assume un ruolo fondamentale
- Paradossalmente con un **waterfall** l'automazione dei test è meno cruciale rispetto **all'Agile**

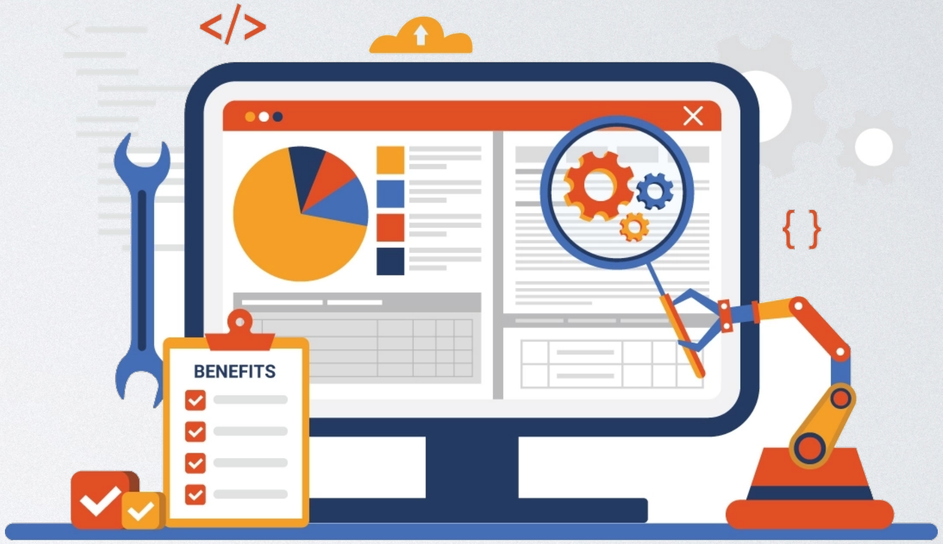


# MANUAL TESTING

- Testare la facilità di utilizzo, estetica o funzionalità differenti rispetto al **design**
- Testare **l'integrazione** tra componenti di vari team differenti (es: PayPal, Google Maps, dispositivi esterni, ecc.)
- Testare gli **happy path cruciali** per la tua app
- Trovare situazioni **imprevedibili** dell'applicazione / usi non convenzionali

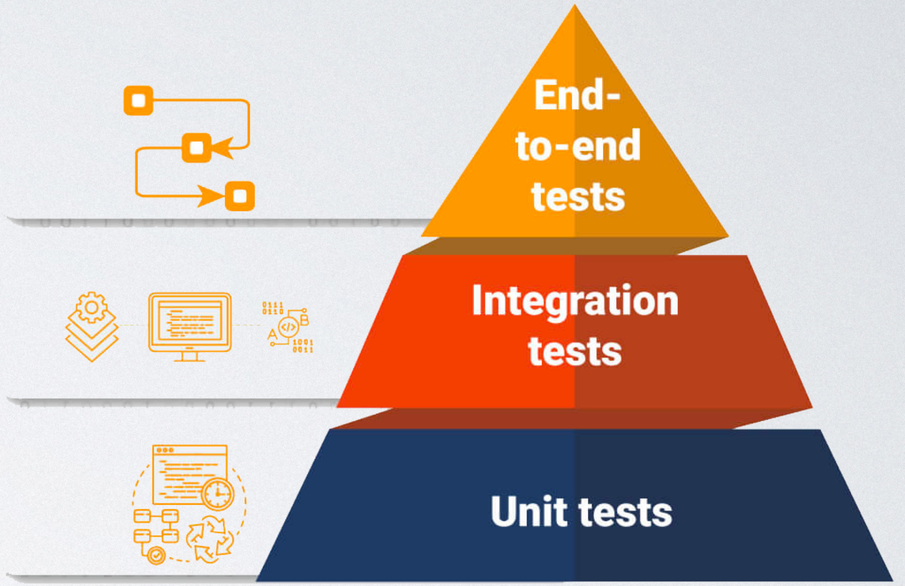
# AUTOMATED TESTING

- Permettono di fare un **design** corretto dell'applicazione pensando prima ai **requisiti** da soddisfare facendo il minimo codice necessario (YAGNI)
- **Accorciano il feedback-loop** ed evitano di mettere in master codice che **rompe** funzionalità esistenti
- Fanno **risparmiare molto tempo di testing** manuale per gli use-case già definiti e ripetitivi
- Verificano i **requisiti di accettazione delle storie** e permettono di non perderli nel tempo / con il turnover di product manager e sviluppatori
- Cruciali per fare **Continuous Integration** e **Continuous Delivery** e mantenere un'alta qualità del codice
- **Start-up** che non hanno un budget per un QA. Spesso non si fanno per inesperienza dei dev.
- Sono generalmente **facili da mantenere** e **descrittivi** del comportamento atteso del software



# TIPI DI TEST

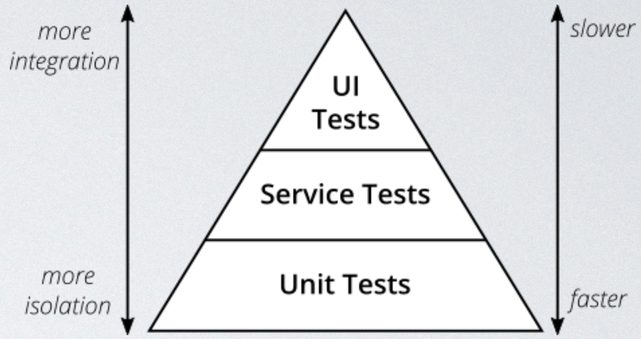
- **Unit-test:** verificano una funzionalità in isolamento. Semplici, veloci. Gli altri oggetti sono mockati.
- **Integration test:** verifica l'integrazione fra vari oggetti o layer. Ci sono vari oggetti reali.
- **End to end test**: fanno un test globale della tua applicazione. Gli oggetti sono per la maggior parte reali.

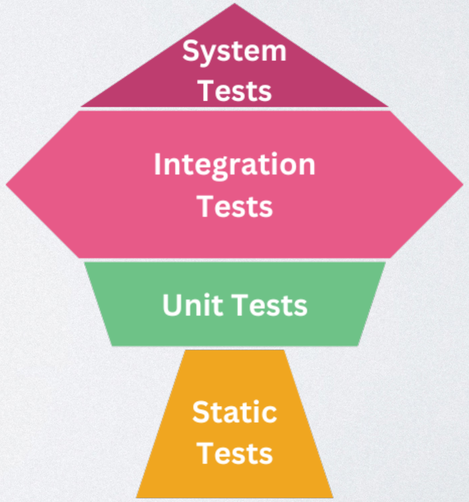


# TEST PYRAMID VS TEST TROPHY

- **Test Pyramid:** più **facile** da mantenere, test più **veloci**. Possibile che alcuni **bug** sfuggano essendo nell'integrazione tra i vari layer. Richiede un software con **dependency injection.**

- **Test Trophy:** più **difficile** da mantenere e capire dove crassa. Test più **lenti.** Viene testata anche l'integrazione. Non serve necessariamente la **dependency injection** (dipende).



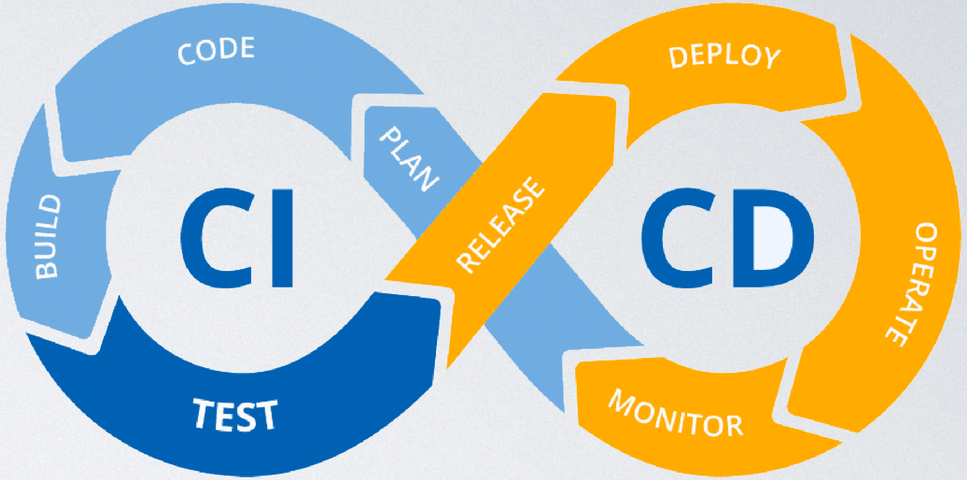

To avoid any confusion, Michael Feathers gives a clear definition of **what is NOT** a unit test.

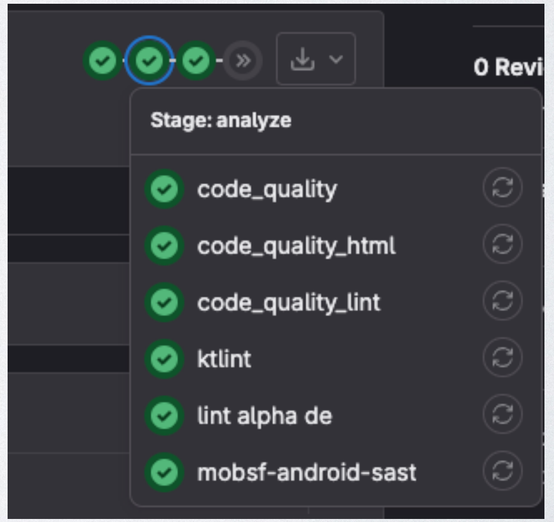
In short, your test is not unit if:

1. it doesn't run fast (< 100ms / test)
2. it talks to the Infrastructure (e.g. a database, the network, the file system, environment variables…)

# CONTINUOUS INTEGRATION & DELIVERY

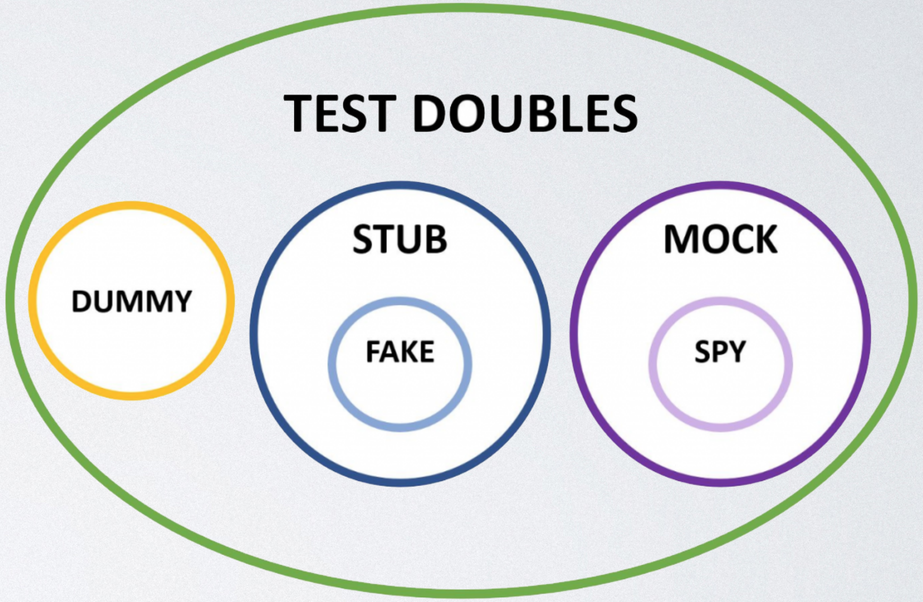
- **Verificare** sia la **build** che i **test** prima di mergiare in master
- Rendere **oggettiva** la validazione del codice e step obbligatorio
- Verifiche **statiche obbligatorie** (lint / ktlint)
- **Deployare** in tempi brevi app per ambienti diversi / con **configurazioni** diverse
- Rendere **sicuro** e automatico il deploy



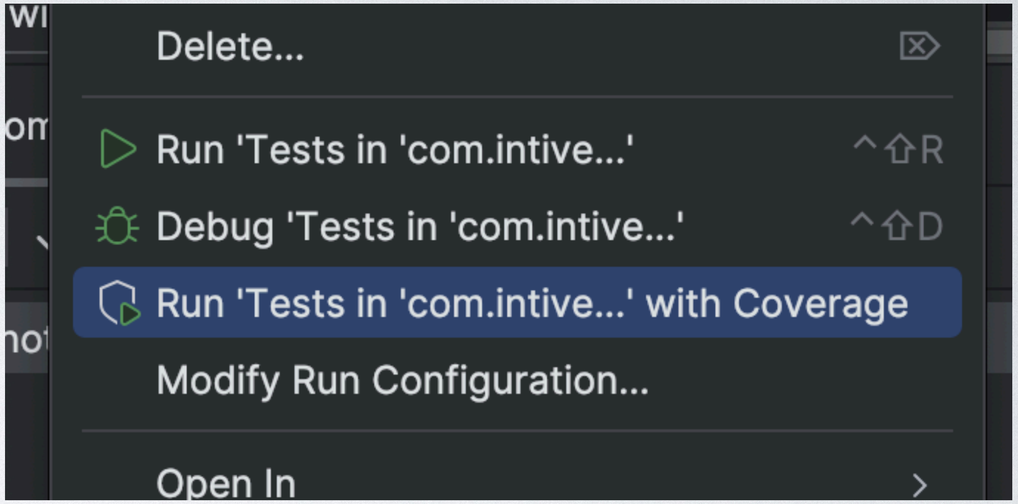
# DUMMY, FAKE, STUB, SPY, MOCK

- **Dummy:** oggetti passati senza implementazione (di solito mock vuoti)
- **Fake:** oggetti con implementazione reale che usano scorciatoie
- **Stub:** oggetti con risposte dirette preconfezionate o nessuna implementazione.
- **Spy:** stub che registrano come vengono chiamati i vari metodi.
- **Mock:** oggetti pre-programmati con delle "expectation" di comportamento



# TEST COVERAGE… A COSA SERVE?

- **Verificare** se ci sono path non colpiti dai test.
- **Riferire al business** quanto codice è stato coperto dai test.
- Non è un parametro fondamentale (se non da un punto di vista contrattuale)




| | Class, % | Method, % | Line, % | Branch, % |
|---|---|---|---|---|
| auth | 50% (104/... | 41% (207/496) | 36% (419/1150) | 28% (119/415) |
| auth0 | 50% (13/26) | 36% (13/36) | 19% (13/66) | 0% (0/11) |
| consents | 47% (8/17) | 46% (23/50) | 40% (45/110) | 7% (2/26) |
| deepLink | 50% (6/12) | 43% (16/37) | 43% (40/93) | 29% (10/34) |
| dialogs | 40% (19/47) | 32% (40/124) | 27% (75/273) | 12% (12/94) |
| email | 55% (10/18) | 38% (14/36) | 37% (33/87) | 21% (6/28) |
| enterOtp | 61% (11/18) | 60% (35/58) | 53% (83/156) | 52% (37/70) |
| main | 60% (17/28) | 39% (23/58) | 33% (48/142) | 41% (26/62) |
| market | 100% (0/0) | 100% (0/0) | 100% (0/0) | 100% (0/0) |
| verificationOtp | 68% (15/22) | 55% (33/60) | 46% (71/154) | 40% (26/64) |
| welcome | 31% (5/16) | 27% (10/37) | 15% (11/69) | 0% (0/26) |

```
41
42      override fun onRetryButtonClick() {
43          loadAccessories()
44      }
45
46      fun onAccessoryClick(accessory: Acce
47          navigator.call(NavigateToPeriphe
48      }
```

# MOCK VS FAKE

```
class UserManagerTest {

    private lateinit var userRepository: UserRepository
    private lateinit var userManager: UserManager

    @Before
    fun setUp() {
        userRepository = mockk()
        userManager = UserManager(userRepository)
    }

    @Test
    fun `getUserName returns user name when user exists`() {
        val userId = 1
        val userName = "John Doe"
        val user = User(userId, userName)
        every { userRepository.getUser(userId) } returns user
        val result = userManager.getUserName(userId)
        assertEquals(userName, result)
        verify(exactly = 1) { userRepository.getUser(userId) }
    }
```

```
interface UserRepository {
    fun save(user: User)
    fun findById(id: Int): User?
}

class FakeUserRepository : UserRepository {
    private val users = mutableMapOf<Int, User>()

    override fun save(user: User) {
        users[user.id] = user
    }

    override fun findById(id: Int): User? = users[id]
}

class User(val id: Int, val name: String)

fun main() {
    val fakeRepository = FakeUserRepository()
    val user = User(1, "John Doe")
    fakeRepository.save(user)
    val retrievedUser = fakeRepository.findById(1)
    retrievedUser shouldBe User(1, "John Doe")
}
```

# SETUP, EXERCISE, VERIFY, TEARDOWN

```
interface UserRepository {
    fun save(user: User)
    fun findById(id: Int): User?
    fun clear()
}

class User(val id: Int, val name: String)

class UserViewModel(private val userRepository: UserRepository) {
    fun saveUser(user: User) {
        userRepository.save(user)
    }

    fun getUserById(id: Int): User? {
        return userRepository.findById(id)
    }
}
```

```
class UserRepositoryTest {
    private val mockRepository: UserRepository = mock()
    private val user = User(1, "John Doe")
    private lateinit var viewModel: UserViewModel

    @Before
    fun setup() {
        viewModel = UserViewModel(mockRepository)
    }

    @After
    fun teardown() {
        mockRepository.clear()
    }

    @Test
    fun `test saveUser`() {
        viewModel.saveUser(user)

        verify(mockRepository).save(user)
    }

    @Test
    fun `test getUserById`() {
        whenever(mockRepository.findById(1)).thenReturn(user)

        val retrievedUser = viewModel.getUserById(1)

        retrievedUser shouldBe user
    }
}
```

# I TEST SULLA UI… COME LI FACCIAMO?

- **Test con JetPack Compose:** quasi mai visto che la logica dovrebbe essere tutta sul viewModel. Al massimo si possono testare le varie condizioni.
- **Test con espresso + ui test:** sono **molto lenti**. Solo in caso non si riesca a fare a meno
- **Test con robolectric:** sono più **veloci** ma più difficili da mantenere (specie con gli XML)

# ESPRESSO, KASPRESSO, TEST CON JETPACK COMPOSE

```
@Test
fun greeterSaysHello() {
    onView(withId(R.id.name_field)).perform(typeText("Steve"))
    onView(withId(R.id.greet_button)).perform(click())
    onView(withText("Hello Steve!")).check(matches(isDisplayed()))
}
```

```
@Test
fun testFirstFeature() {
    mainScreen {
        toFirstFeatureButton {
            isVisible()
            click()
        }
    }
}
```

```
@Test  Filippo De Pretto +1
fun `Intro text is 'Add, delete, or edit as needed' WHEN floorPlansLimit is over 1`() {
    rule.setContent {
        VrxFloorPlanListContent(
            listOfFloorplans,
            floorPlansLimit: 5,
            addFloorPlanButtonState = Enabled,
            isMultipleFloorPlanAvailable = true,
            actions = VrxFloorPlanActions({}, {}, {}, {}, {}, {}),
        )
    }

    rule.onNodeWithText(editYourFloorPlan).assertIsNotDisplayed()
    rule.onNodeWithText(addDeleteEditText, substring = true).assertIsDisplayed().assertIsEnabled()
}
```

- I test con **espresso** sono molto più **verbosi**. Kaspresso con **Kakao** semplifica molto la DSL ma bisogna conoscerla.
- I test di **Jetpack Compose** sono (quasi) **fini a se stessi**

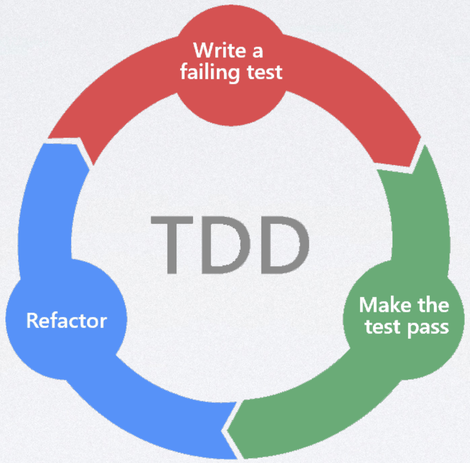
# TEST CON I FLOW, COROUTINE E TURBINE

```
class UserViewModel(private val userRepository: UserRepository) : ViewModel() {
    private val _userFlow = MutableStateFlow<User?>(null)
    val userFlow: StateFlow<User?> get() = _userFlow

    fun fetchUser(userId: Int) {
        viewModelScope.launch {
            val user = userRepository.getUserById(userId)
            _userFlow.value = user
        }
    }
}

interface UserRepository {
    suspend fun getUserById(userId: Int): User?
}

data class User(val id: Int, val name: String)
```

```
class UserViewModelTest {
    private val mockRepository: UserRepository = mock()
    private val user = User(1, "John Doe")
    private lateinit var viewModel: UserViewModel
    private val testDispatcher = UnconfinedTestDispatcher()

    @Before
    fun setup() {
        Dispatchers.setMain(testDispatcher)
        viewModel = UserViewModel(mockRepository)
    }

    @After
    fun teardown() {
        Dispatchers.resetMain()
    }

    @Test
    fun `test getUserById`() = runTest {
        whenever(mockRepository.findById(1)).thenReturn(user)

        viewModel.userFlow.test {
            viewModel.fetchUser(1)

            val retrievedUser = awaitItem()
            retrievedUser shouldBe user
        }
    }
}
```

# TEST DRIVEN DEVELOPMENT & ANDROID

- **Comprendo** il dominio e cosa mi viene chiesto
- **Scrivo un test** per verificare quel requisito utente
- Il test è **rosso** perché la funzionalità non è ancora implementata
- Scrivo la **minima implementazione** per far funzionare il test
- Il test è **verde**
- A questo punto se voglio aggiungere funzionalità, **scrivo un altro test** che deve fallire
- Altrimenti faccio **refactor** e miglioro il codice
- **KISS** (Keep It Simple Stupid) & **YAGNI** (You aren't Gonna Need It)




```
fun syncSpotlights() = viewModelScope.launch {
    spotlightRepository.sync()
}

fun showSpotlights() = viewModelScope.launch {
    analytics.logEvent(OPEN_PUSH, null)
    events.emit(ShowSpotlights)
}

fun fetchNotifications() = viewModelScope.launch {
    reminderRepository.fetchReminders()
}

fun checkFeatureFlag() {
    if (isFeatureFlagEnabled && developerRepository.isNewBuild) {
        openFeatureFlag()
        developerRepository.updateBuild()
    }
}

fun openFeatureFlag() = viewModelScope.launch {
    events.emit(OpenFeatureFlag)
}
```

# TEST FACTORY CON JUNIT5

- Quando abbiamo molti **parametri** da andare a testare, una struttura comoda sono le **test factory**. Questo ci permette di definire la struttura su cui lanciare vari casi d'uso.

- Voglio **evitare** il più possibile **if/else** nei test anche in questo caso

```
@TestFactory
fun `Test Device Pairing Errors`() =
    listOf(
        Pair(null, Regular) to DeviceNotInPairingMode,
        Pair(null, NonConnectible) to TooManyDevicesConnected,
        Pair("serial123", Regular) to DeviceNotInPairingMode,
        Pair("serial123", NonConnectible) to TooManyDevicesConnected,
    ).map { (input, error) ->
        val (serialToPair, mode) = input
        dynamicTest("When serial is $serialToPair and mode is $mode, " +
                    "it should return error $error") {
            every { bleScanResult.mode } returns mode

            val result = tested.execute(serialToPair).test()

            result shouldEqualError error
        }
    }
```

# LEGACY CODE & REFACTORING

> *“Legacy Code is code without tests”*

- Prima **aggiungi i test** poi fai refactoring
- Identifica i **punti di contatto**, rompi le dipendenze, scrivi i test, fai le tue modifiche.
- **UI Test / Snapshot test** per verificare il comportamento più che le singole unità
- Evita di chiamare direttamente le **librerie** ma, preferibilmente, usa delle astrazioni

# LEGACY CODE: SPROUT EXAMPLE

**Sprout technique:** se devi fare una modifica a del codice legacy, crea un metodo / oggetto nuovo, testalo e innestalo in quello vecchio.

```kotlin
// LegacyCalculator.kt
package com.example.legacyapp

import android.content.Context
import android.widget.Toast

class LegacyCalculator(private val context: Context) {

    fun add(a: Int, b: Int): Int {
        val result = a + b
        Toast.makeText(context, "Result: $result", Toast.LENGTH_SHORT).show()
        return result
    }

    fun multiply(a: Int, b: Int): Int {
        val result = a * b
        Toast.makeText(context, "Result: $result", Toast.LENGTH_SHORT).show()
        return result
    }
}
```

```kotlin
class NewCalculatorTest {

    private val newCalculator = NewCalculator()

    @Test
    fun testSubtract() {
        // Given
        val a = 10
        val b = 5
        val expectedResult = 5

        // When
        val result = newCalculator.subtract(a, b)

        // Then
        assertEquals(expectedResult, result)
    }
}
```

```kotlin
class NewCalculator {

    fun subtract(a: Int, b: Int): Int {
        return a - b
    }
}
```

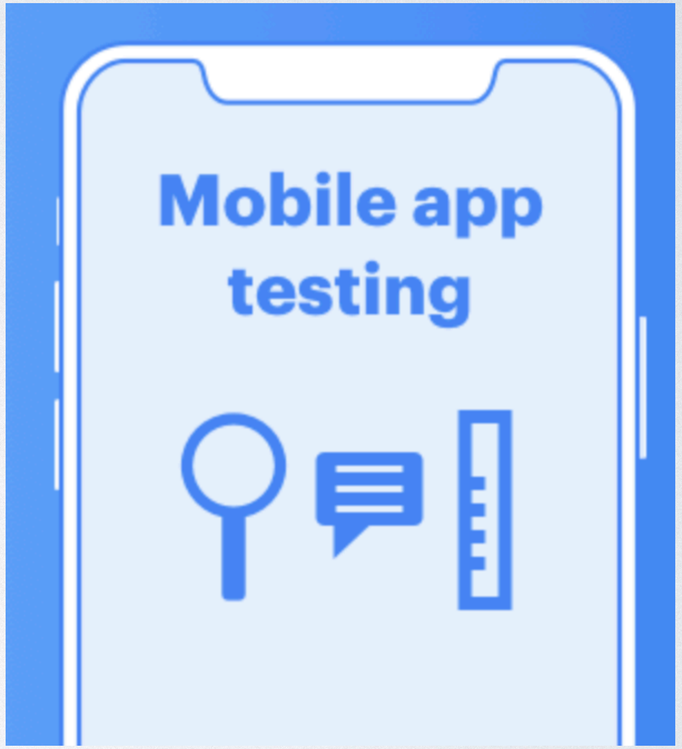
# LEGACY CODE: WRAP TECHNIQUE

**Wrap technique:** puoi wrappare un vecchio metodo creando un oggetto che lo contiene e chiamare un oggetto nuovo che rimanda al vecchio. Per poi mockare il vecchio.

```
@Before
fun setUp() {
    context = mock(Context::class.java)
    legacyCalculator = LegacyCalculator(context)
    calculatorWrapper = CalculatorWrapper(legacyCalculator)
}

@Test
fun testAdd() {
    // Given
    val a = 5
    val b = 10
    val expectedResult = 15

    // When
    val result = calculatorWrapper.add(a, b)

    // Then
    assertEquals(expectedResult, result)
    verify(context).getString(eq(R.string.result), eq(expectedResult))
}
```

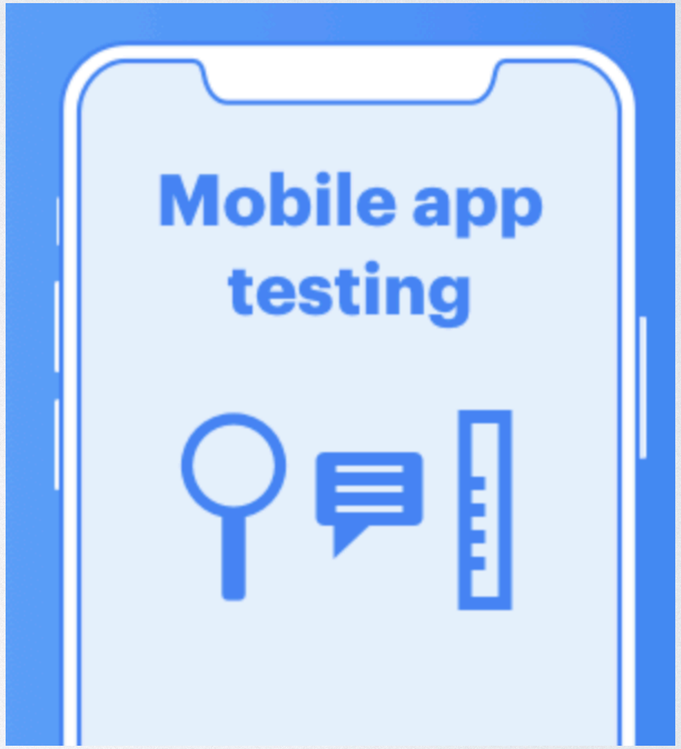
# ERRORI COMUNI (1)

- **Non sapere** cosa stai testando / testare a caso
- Testare il **context**, Activity o Fragment in modo unitario
- Fare solo **test UI**
- Testare cose **inutili / tautologiche**
- Test con uno **stato comune** che si rompono in modo difficile da prevedere



# ERRORI COMUNI (2)

- Mockare **troppo** al punto di non testare nulla
- Poter **rompere master / la CI** o non usarla proprio
- Dover **lanciare i test a mano** prima di mergiare in master
- **Cambiare** il codice di **produzione** per far girare i test / testare i **metodi privati**
- **Testare più cose** nello stesso test (va bene con gli UI Test)



# RIFERIMENTI PER APPROFONDIMENTI

- Repo di XPeppers: **https://github.com/xpeppers/starway-to-orione**
- XP Explained
- Growing Object-Oriented Software, Guided by Tests
- Test Driven Development: By Example
- https://www.kodeco.com/29746623-test-driven-development-in-android

**GRAZIE PER L'ATTENZIONE :-)**

Ore 14:30 parte pratica

Keep in touch: https://www.linkedin.com/in/filnik/