

# Writing tests that stand the test of time



Segun Famisa  
Android GDE



segunfamisa

[segunfamisa.com](http://segunfamisa.com)



# Outline

Introduction to TDD

Challenges with TDD

Testing tools in practice

Writing maintainable tests

Recap

# Introduction

The background of the slide features a dark purple gradient. Overlaid on this is a faint, glowing purple network graph. The graph consists of numerous small, semi-transparent purple dots of varying sizes scattered across the frame, connected by thin, translucent purple lines that form a complex web-like structure.

# So, what's Test Driven Development?

“

TDD is a software development process that relies on the repetition of a very short development cycle: requirements are turned into very specific test cases, then the software is improved to pass the new tests, only

TDD is a software development process that relies on the repetition of a very short development cycle: requirements are turned into very specific test cases, then the software is improved to pass the new tests, only

TDD is a software development process that relies on the repetition of a very short development cycle: requirements are turned into very specific test cases, then the software is improved to pass the new tests, only

The background of the slide features a dark purple gradient with a subtle, glowing purple network of dots and lines. This network is more prominent in the top right corner and has a few bright highlights.

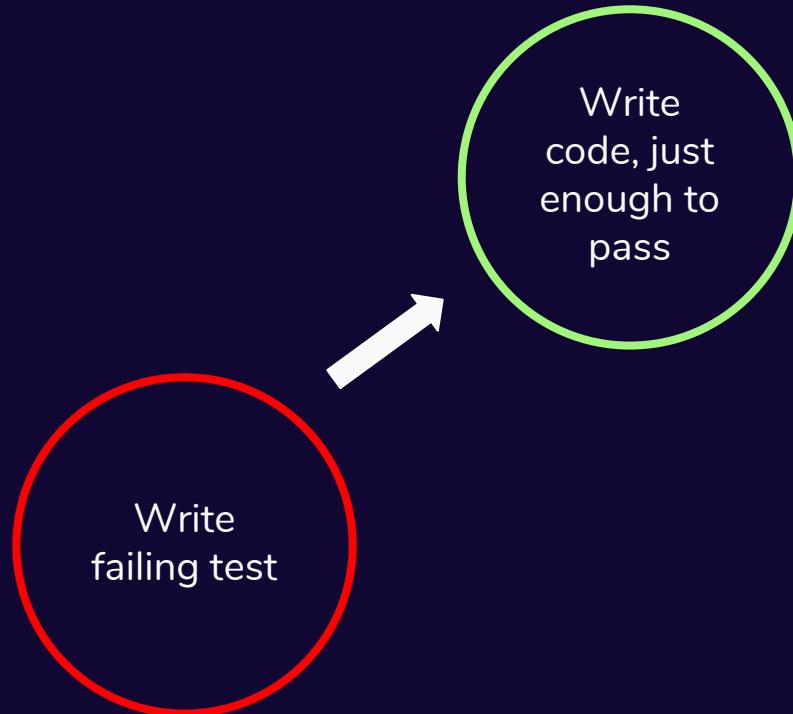
# How to do TDD?

**Red - Green - Refactor**

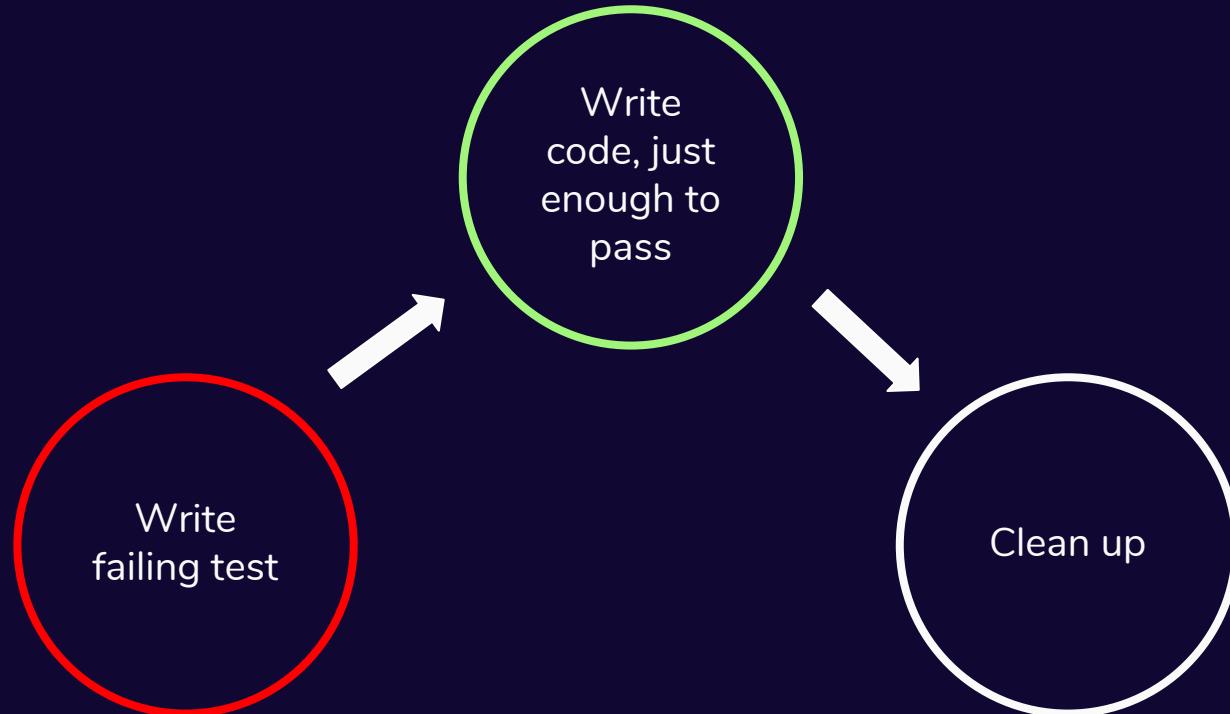
# Red - Green - Refactor



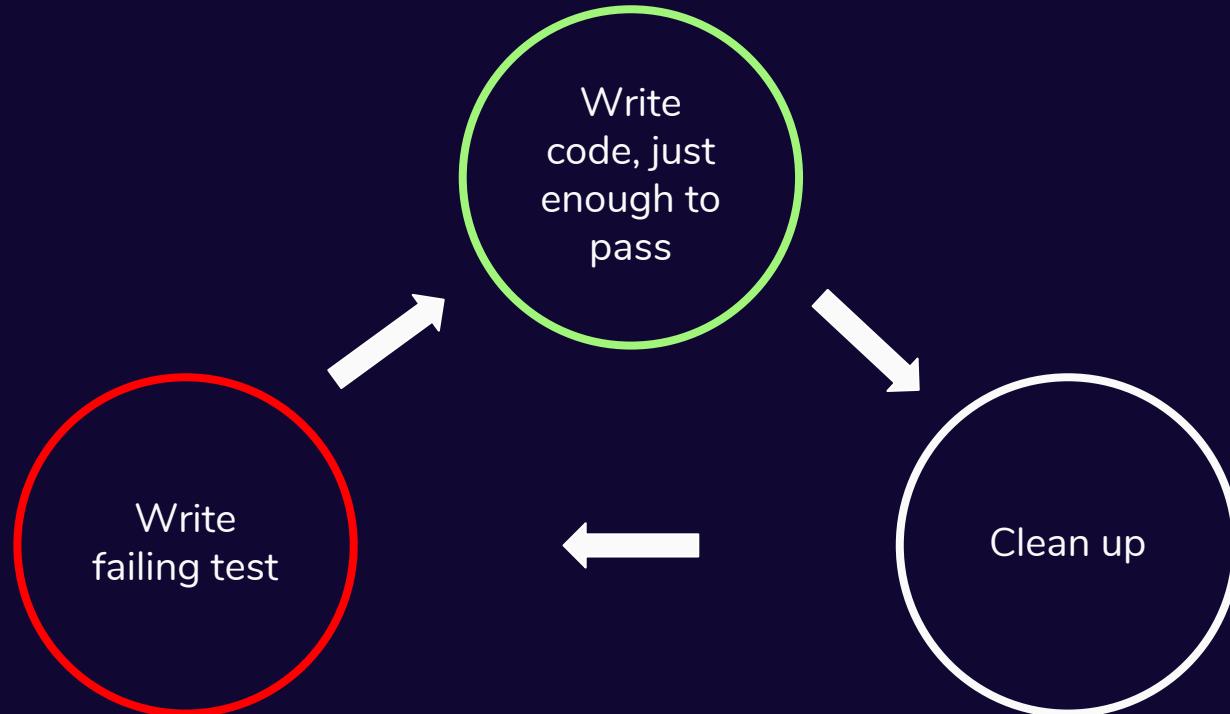
# Red - Green - Refactor



# Red - Green - Refactor



# Red - Green - Refactor



The background of the slide features a dark purple gradient with a subtle, glowing purple network pattern. This pattern consists of numerous small, semi-transparent purple dots connected by thin, translucent purple lines, creating a sense of depth and connectivity.

# Why do we need tests?

# Why do we need tests?

- Quick feedback about bugs/errors

# Why do we need tests?

- Quick feedback about bugs/errors
- Good code design

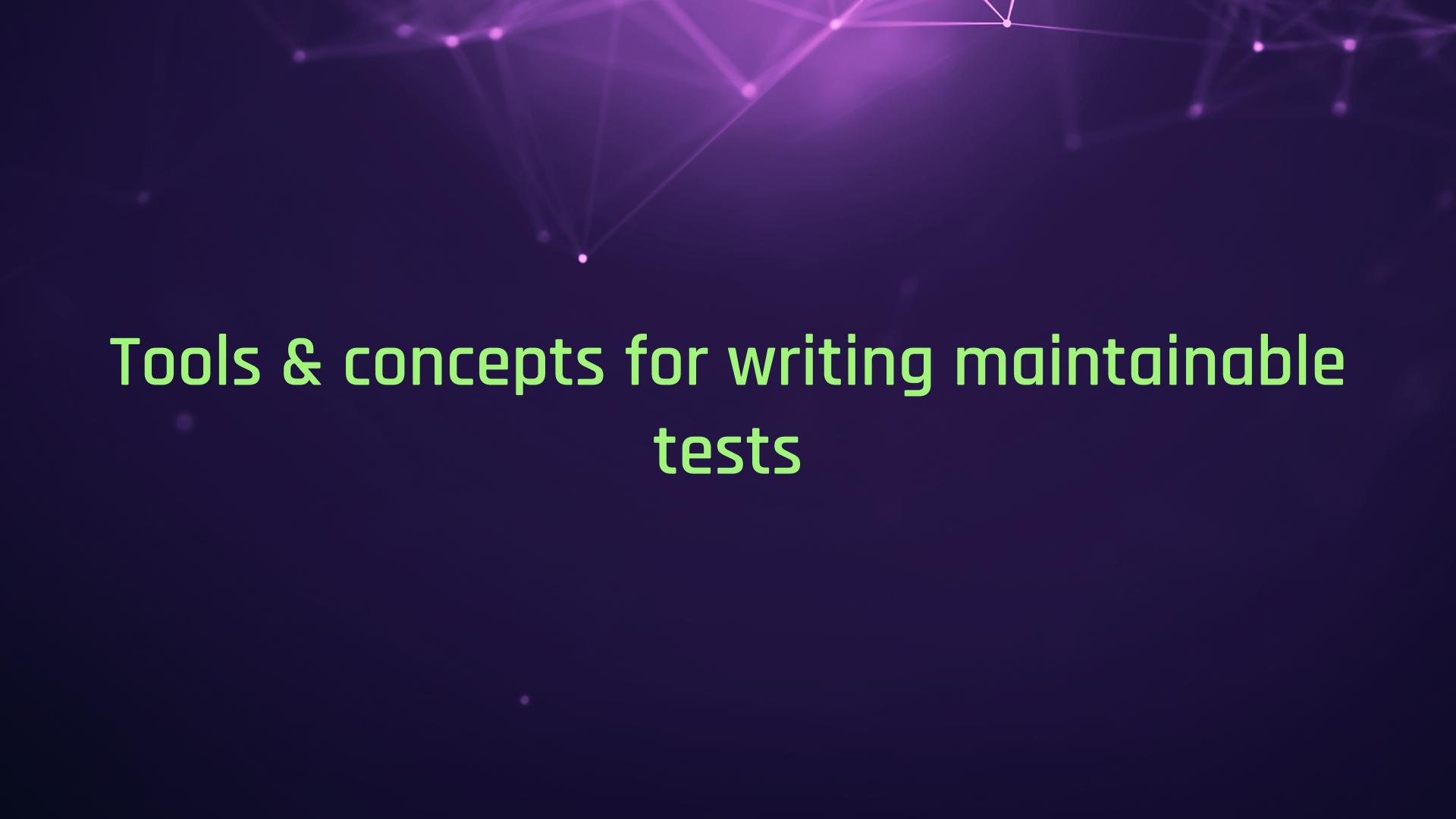
# Why do we need tests?

- Quick feedback about bugs/errors
- Good code design
- Documentation for code behavior

# Why do we need tests?

- Quick feedback about bugs/errors
- Good code design
- Documentation for code behavior
- Confident refactoring

# Challenges with TDD

The background of the slide features a dark purple gradient with a subtle, glowing purple network of dots and lines that suggests a complex system or data structure.

# Tools & concepts for writing maintainable tests

# Test doubles

# Test doubles

Just like stunt doubles



<https://people.com/movies/actors-and-their-stunt-doubles-photos>

# Test doubles - dummies

# Test doubles - dummies

- Dummies are like placeholders. Just to fill in parameters.

# Test doubles - stubs

## Test doubles - stubs

- Objects that return predefined data
- They usually don't hold state/respond to other actions besides the one they are created for

# Test doubles - stubs

```
interface IUserRepository { . . . }
```

# Test doubles - stubs

```
interface IUserRepository { ... }

...

class UserRepository(private val userDao: UserDao) : IUserRepository {

    override fun getUser(userId: Long): User {
        return userDao.findById(userId)
    }
}
```

# Test doubles - stubs

```
interface IUserRepository { ... }

...
class UserRepositoryStub() : IUserRepository {

    override fun getUser(userId: Long): User {
        return User(userId = 1, email = "sf@sf.com")
    }
}
```

Stub returns a preconfigured user

# Test doubles - fakes

## Test doubles - fakes

- Similar to stubs, slightly more realistic
- Contain working implementation, but different from real version
- Typically models the behavior of the real class

# Test doubles - fakes

```
interface UserDao { ... }

...
class FakeUserDao() : UserDao {

    val users = mutableListOf<User>()

    override fun insert(user: User) {
        users.add(user)
    }

    override fun findById(userId: Long): User {
        return users.find { it.userId == userId } ?: throw Exception("user not found")
    }
}
```

# Test doubles - fakes

```
interface UserDao { ... }

...
class FakeUserDao() : UserDao {

    val users = mutableListOf<User>()

    override fun insert(user: User) {
        users.add(user)
    }

    override fun findById(userId: Long): User {
        return users.find { it.userId == userId } ?: throw Exception("user not found")
    }
}
```

Fake dao uses a list  
instead of a db

# Test doubles - fakes

```
interface UserDao { ... }

...
class FakeUserDao() : UserDao {

    val users = mutableListOf<User>()

    override fun insert(user: User) {
        users.add(user)
    }

    override fun findById(userId: Long): User {
        return users.find { it.userId == userId }
            ?: throw Exception("user not found")
    }
}
```

Fake dao supports the same operations

# Test doubles - mocks

# Test doubles - mocks

- Objects pre-programmed with expected outputs for given inputs
- Ability to record method calls and verify them
- Throw exceptions if wanted method is not called

# Test doubles - mocks

```
@Test  
fun userShouldBeReturnedFromDao() {  
  
    val dao: UserDao = mock()  
    whenever(dao.getUser(userId = 5)).thenReturn(User(5, "sf@sf.com"))  
  
    ...  
  
}
```

# Test doubles - mocks

```
@Test  
fun userShouldBeReturnedFromDao() {  
  
    val dao: UserDao = mock()  
    whenever(dao.getUser(userId = 5)).thenReturn(User(5, "sf@sf.com"))  
  
    ...  
  
}
```

Mock pre-programmed with  
input/output

## Test doubles - mocks

```
// SettingsPresenterTest.kt

@Test
fun clickingIconShouldOpenProfileScreen() {
    ...
    val view: SettingsContract.View = mock()
    val presenter = SettingsPresenter(view, userRepo)
    presenter.profileIconClicked()
    verify(view).openProfileScreen()
}
```

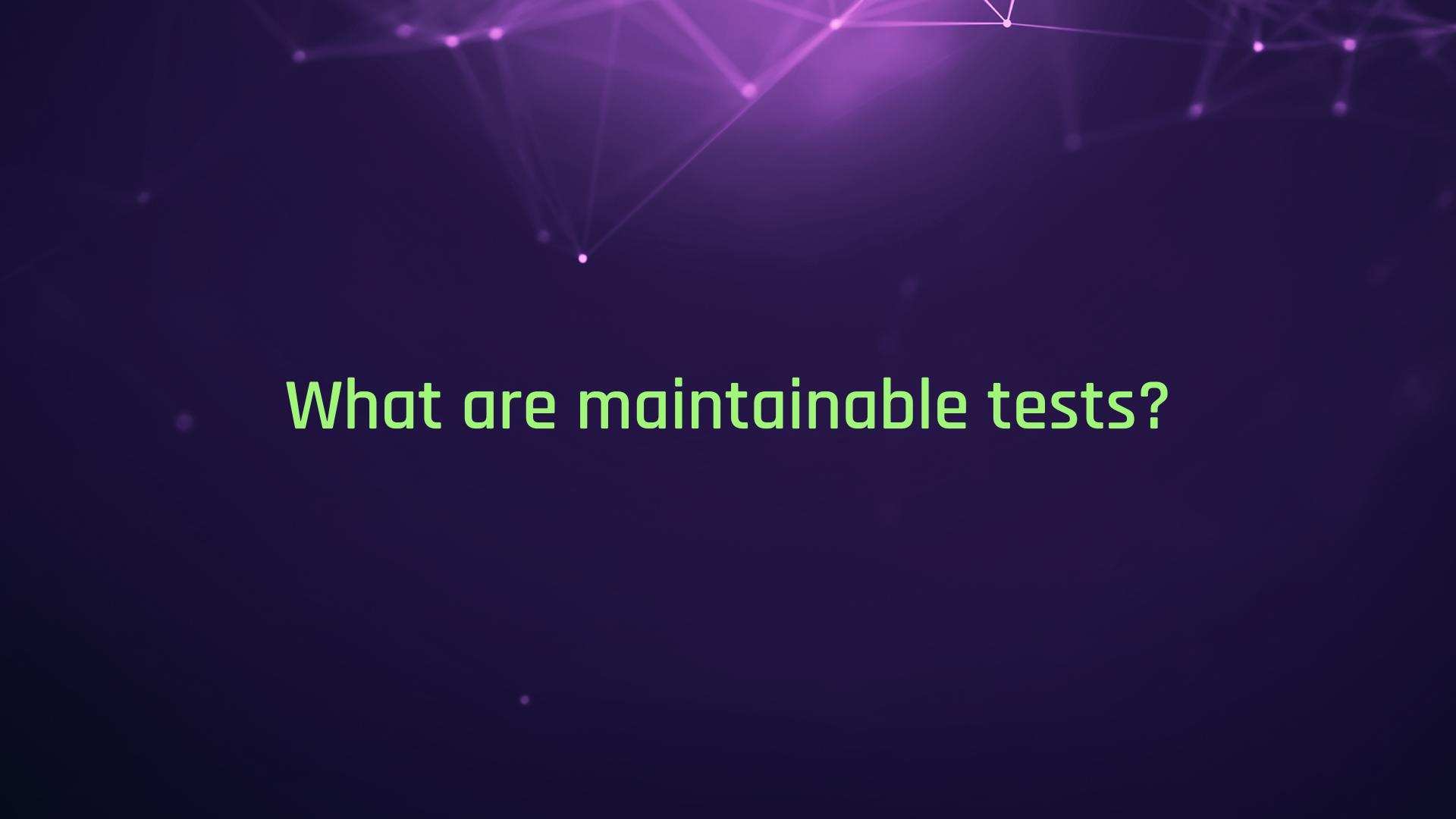
# Test doubles - mocks

```
// SettingsPresenterTest.kt

@Test
fun clickingIconShouldOpenProfileScreen() {
    ...
    val view: SettingsContract.View = mock()
    val presenter = SettingsPresenter(view, userRepo)
    presenter.profileIconClicked()

    verify(view).openProfileScreen()
}
```

Ability to verify interactions

The background of the slide features a dark purple gradient with a subtle, glowing purple network of dots and lines. This network is more concentrated in the upper right quadrant, creating a sense of depth and connectivity.

# What are maintainable tests?

# What are maintainable tests?

Tests are maintainable when:

- Old tests do not break often

# What are maintainable tests?

Tests are maintainable when:

- Old tests do not break often
- Old tests are easy to update

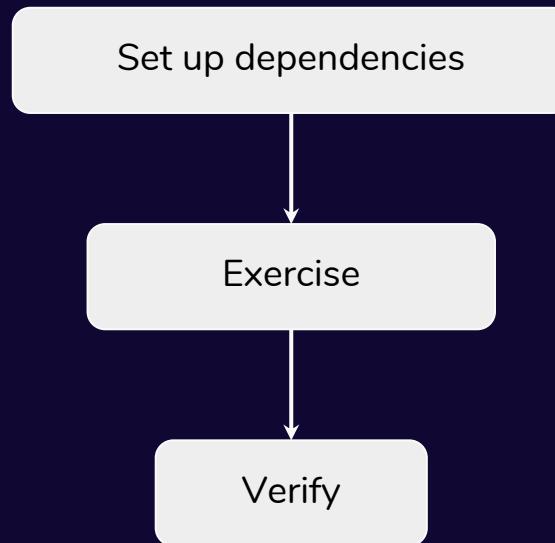
# What are maintainable tests?

Tests are maintainable when:

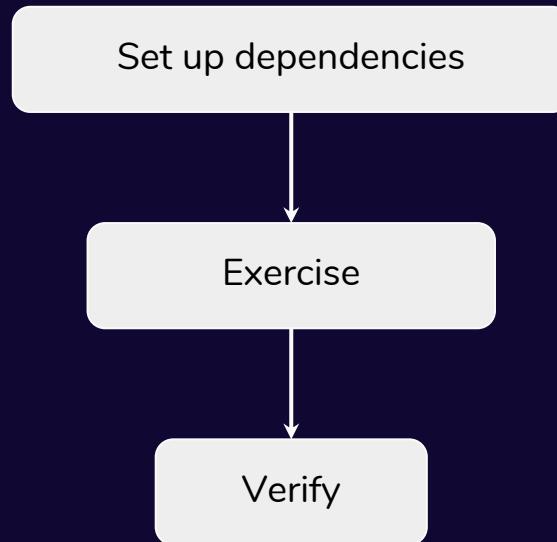
- Old tests do not break often
- Old tests are easy to update
- Easy to add new tests

# Writing maintainable tests

# 1. Use a good test specification system



# 1. Use a good test specification system



Also known as

**Arrange - Act - Assert**

Or

**Given - When - Then**

# 1. Use a good test specification system

```
@Test  
fun scenarioX() {  
  
    // Given the dependencies/behavior  
  
    // When we act on the scenario  
  
    // Then assert that expected behavior happens  
  
}
```

## 2. Test behavior, not implementation details

## 2. Test behavior, not implementation details

- For methods that return value, you should care only about the output, not how it was calculated.

## 2. Test behavior, not implementation details

```
@Test
fun `get user details from cache if available`() {
    ...
    val userRepo = UserRepository(cacheSource, networkSource)

    // given that a user exists in cache
    whenever(cacheSource.getUser(5)).thenReturn(User(5, "sf@sf.com"))

    // when we get user from repository
    val user = userRepo.getUser(userId = 5)

    // then verify that the cache source was called
    verify(cacheSource).getUser(5)
}
```

## 2. Test behavior, not implementation details

```
@Test  
fun `get user details from cache if available`() {  
    ...  
    val userRepo = UserRepository(cacheSource, networkSource)  
  
    // given that a user exists in cache  
    whenever(cacheSource.getUser(5)).thenReturn(User(5, "sf@sf.com"))  
  
    // when we get user from repository  
    val user = userRepo.getUser(userId = 5)  
  
    // then verify that the cache source was called  
    verify(cacheSource).getUser(5)  
}
```

This tests implementation details

## 2. Test behavior, not implementation details

```
@Test
fun `get user details from cache if available`() {
    ...
    // given that a user exists in cache
    val cachedUser = User(5, "sf@sf.com")
    whenever(cacheSource.getUser(5)).thenReturn(cachedUser)

    // when we get user from repository
    val returnedUser = userRepo.getUser(userId = 5)

    // then verify that the returned user is the one from cache
    assertEquals(cachedUser, returnedUser)
}
```

## 2. Test behavior, not implementation details

```
@Test  
fun `get user details from cache if available`() {  
    ...  
    // given that a user exists in cache  
    val cachedUser = User(5, "sf@sf.com")  
    whenever(cacheSource.getUser(5)).thenReturn(cachedUser)  
  
    // when we get user from repository  
    val returnedUser = userRepo.getUser(user)  
  
    // then verify that the returned user is the one from cache  
    assertEquals(cachedUser, returnedUser)  
}
```

This tests general behavior of this repository in this scenario.

## 2. Test behavior, not implementation details

- For methods that return value, one should care only about the output, not how it was calculated.
- For methods that do not return any value, verify interactions with dependencies

## 2. Test behavior, not implementation details

- For methods that return value, one should care only about the output, not how it was calculated.
- For methods that do not return any value, verify interactions with dependencies
- Be careful about overusing mocks.

### 3. Assert/verify only one thing per test

### 3. Assert/verify only one thing per test

In most cases, only one assert / verify should be done in each test.

### 3. Assert/verify only one thing per test

In most cases, only one assert / verify should be done in each test.

A test should fail for only 1 reason

### 3. Assert/verify only one thing per test

```
@Test  
fun `enabling setting updates preference and sends tracking`() {  
    ...  
    // when user enables the setting  
    viewModel.enableSetting()  
  
    // then verify that we set preference  
    verify(userPreference).enableSetting()  
  
    // then verify that we send tracking  
    verify(trackingUtils).trackUserEnabledSetting()  
}
```

### 3. Assert/verify only one thing per test

```
@Test  
fun `enabling setting updates preference and sends tracking`() {  
    ...  
    // when user enables the setting  
    viewModel.enableSetting()  
  
    // then verify that we set preference  
    verify(userPreference).enableSetting()  
  
    // then verify that we send tracking  
    verify(trackingUtils).trackUserEnabledSetting()  
}
```

The use of “and” suggests  
that the test is testing more  
than one thing

### 3. Assert/verify only one thing per test

```
@Test  
fun `enabling setting updates preference`() {  
    ...  
    // then verify that we set preference  
    verify(userPreference).enableSetting()  
}
```

```
@Test  
fun `enabling setting posts tracking`() {  
    ...  
    // then verify that we post tracking  
    verify(trackingUtils).trackUserEnabledSetting()  
}
```



## 4. Use descriptive test names

## 4. Use descriptive test names

From the test name, we should be able to tell why the test failed.

## 4. Use descriptive test names

From the test name, we should be able to tell why the test failed.

```
@Test  
fun `search field is updated correctly when user has search history`() {  
    ...  
}
```

## 4. Use descriptive test names

From the test name, we should be able to tell why the test failed.

```
@Test  
fun `search field is updated correctly when user has search history`() {  
    ...  
}
```



## 4. Use descriptive test names

From the test name, we should be able to tell why the test failed.

```
@Test  
fun `search field is updated correctly when user has search history`() {  
    ...  
}
```

```
@Test  
fun `search field is updated with last search when user has search history`() {  
    ...  
}
```

## 4. Use descriptive test names

From the test name, we should be able to tell why the test failed.

```
@Test  
fun `search field is updated correctly when user has search history`() {  
    ...  
}
```

```
@Test  
fun `search field is updated with last search when user has search history`() {  
    ...  
}
```



## 4. Use descriptive test names

Kotlin allows us to use to write test function names with spaces

```
@Test  
fun `welcome dialog should be shown on first log in`() {  
    // test goes here  
}
```

## 4. Use descriptive test names

JUnit 5 allows to specify a custom display name for the test

```
@Test  
@DisplayName("welcome dialog should be shown on first log in")  
void showWelcomeDialogOnFirstLogin() {  
    // test goes here  
}
```

## 5. More tips

- Avoid logic in your tests -> if/else, loops, etc.
- Avoid abstractions in tests
- Be generous with comments
- Use parameterized tests

# Resources

- <https://martinfowler.com/articles/mocksArentStubs.html>
- <http://xunitpatterns.com/>
- <https://testing.googleblog.com/search/label/TotT>
- <https://mtlynch.io/good-developers-bad-tests/>
- <https://jeroenmols.com/blog/2018/12/06/fixthetest/>

# Thank you!



@segunfamisa