Beyond the Mockery

Ash Davies

ashdavies.dev | ashdavies@androiddev.social

Legacy (adj.)

Denoting or relating to software or hardware that has been superseded but is difficult to replace because of its wide use.

Change

Testing

Testing: Awareness 👀

Testing: Architecture 

Testing: Confidence 💪

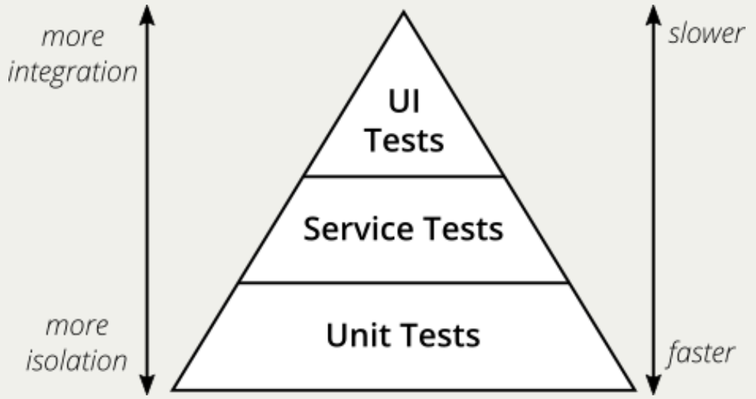
Testing: Documentation 

Testing: Stability 💯

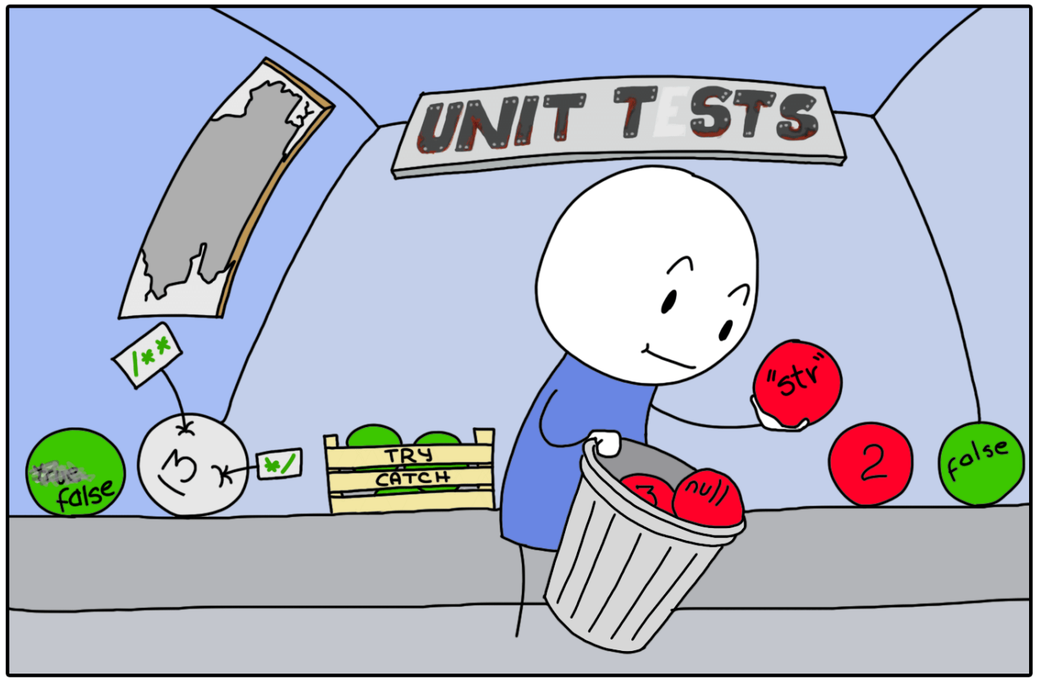
Testing

Testing

- Unit
- Instrumentation
- Integration
- End-to-End
- Monkey
- Smoke



FIXING UNIT TESTS



MONKEYUSER.COM

Anti-Patterns and Code Smell

Extreme Programming 🤘

Test-Driven Development

Debugging is like being the detective in a crime movie where you are also the murderer.

— Filipe Fortes

Kotlin (noun)

Awesome.

Kotlin: Mutability

Risks of Mutation

Kotlin: Mutability

Risks of Mutation

```
fun sumAbsolute(list: MutableList<Int>): Int {
  for (i in list.indices) list[i] = abs(list[i])
  return list.sum()
}
```

Kotlin: Mutability

Risks of Mutation

```
private val GROUNDHOG_DAY = TODO("java.util.Date()")
fun startOfSpring(): java.util.Date = GROUNDHOG_DAY

val partyDate = startOfSpring()
partyDate.month = partyDate.month + 1

// Date is mutable 🤦
```

Kotlin: Mutability

Shared Mutable State ⚠️

Kotlin: Immutability

Unidirectional Data Flow 🏆

Kotlin: Immutability

Collections

```
fun List<T>.toMutableList(): MutableList<T>

fun Map<K, V>.toMutableMap(): MutableMap<K, V>

fun Set<T>.toMutableSet(): MutableSet<T>
```

Kotlin: Immutability

IntelliJ IDEA

```
val immutableValue = true
var mutableValue = false
```

Kotlin: Immutability

Final Concretions 

By default, Kotlin classes are final – they can't be inherited

Kotlin: Immutability

Final Concretions 

```
open class Base // Class is open for inheritance
```

Kotlin: Immutability

All-Open Compiler Plugin

org.jetbrains.kotlin.plugin.allopen

Anti-Patterns and Code Smell

Refactoring: Dependencies

Refactoring: Dependencies

```
==========================================================
diff --git a/CoffeeMaker.kt b/CoffeeMaker.kt

- internal class CoffeeMaker {
-   private val heater: Heater = ElectricHeater()
-   private val pump: Pump = Thermosiphon(heater)
- }

+ internal class CoffeeMaker(
+   private val heater: Heater,
+   private val pump: Pump,
+ )
```

Refactoring: Dependencies

```
==========================================================================
diff --git a/CoffeeMaker.kt b/CoffeeMaker.kt

+ internal class CoffeeMaker(
+   private val heater: Heater,
-   private val thermosiphon: Thermosiphon,
+   private val pump: Pump,
+ )
+
+ internal interface Pump {
+   fun pump()
+ }
+
- internal class Thermosiphon {
+ internal class Thermosiphon : Pump {
```

Testing: Dependencies

Dependency Injection

```
internal class CoffeeMaker(
  private val heater: Heater,
  private val pump: Pump,
)
```

Testing: Dependencies

Dependency Injection

```
val heater = NuclearFusionHeater() // Expensive!

val maker = CoffeeMaker(
  pump = Thermosiphon(heater),
  heater = heater,
)

assertTrue(maker.brew())
```

Testing: Dependencies

Dependency Injection

```
val heater = DiskCachedHeater() // Stateful!

val maker = CoffeeMaker(
  pump = Thermosiphon(heater),
  heater = heater,
)

assertTrue(maker.brew())
```

Testing: Dependencies

Dependency Injection

```
val heater = UnbalancedHeater() // Error prone!

val maker = CoffeeMaker(
  pump = Thermosiphon(heater),
  heater = heater,
)

assertTrue(maker.brew())
```

Testing: Dependencies

Test Doubles

Testing: Dependencies

Mocks

Testing: Dependencies

Mockito

"Tasty mocking framework for unit tests in Java".

Testing: Mocks

```
val heater = mock<Heater>()
val pump = mock<Pump>()

val maker = CoffeeMaker(
  heater = heater,
  pump = pump,
)

assertTrue(maker.brew())
```

Testing: Mocks

```
val heater = mock<Heater>()
val pump = mock<Pump>()

val maker = CoffeeMaker(
  heater = heater,
  pump = pump,
)

assertTrue(maker.brew()) // ⚠ Fails!
```

Testing: Mocks

```
val heater = mock<Heater> {
  on { isHeating } doAnswer { true }
}

val pump = mock<Pump> {
  on { pump() } doAnswer { true }
}

val maker = CoffeeMaker(
  heater = heater,
  pump = pump,
)

assertTrue(maker.brew())
```

Testing: Mocks

javadoc.io/doc/
org.mockito/mockito-core/latest/
org/mockito/Mockito.html

Testing: Mocks

Footguns 🦶🔫

Testing: Mocks

Accidental Invocation

```
val heater = mock<Heater> {
  on { isHeating } doAnswer { true } // Actual invocation!
}
```

Testing: Mocks

Accidental Invocation

```
spy(emptyList<String>()) {
  on { get(0) } doAnswer { "foo" } // throws IndexOutOfBoundsException
}
```

Testing: Mocks

API Sensitivity

```
internal interface Heater {
  val isHeating: Boolean
}

val heater = mock<Heater> {
  on { isHeating } doAnswer { true }
}
```

Testing: Mocks

API Sensitivity

```
 internal interface Heater {
+   fun <T : Any> heat(body: () -> T): T
    val isHeating: Boolean
 }

 val heater = mock<Heater> {
    on { isHeating } doAnswer { true }
 }
```

Testing: Mocks

API Sensitivity

```
internal interface CoffeeDistributor {
  fun announce(vararg name: String): Boolean
}

val mockDistributor = mock<CoffeeDistributor> {
  on { announce(any(), any()) } doReturn true
}

val announced = mockDistributor.announce(
  "Steve", "Roger", "Stan",
)

assertTrue(announced) // False: We only stubbed two names!
```

Testing: Mocks

Default Answers

```
val heater: Heater = mock() // No default answer
val isHeating: Boolean = heater.isHeating // Null
```

Testing: Mocks

Default Answers

```
val heater: Heater = mock(defaultAnswer = RETURNS_SMART_NULLS)
val isHeating: Boolean = heater.isHeating // false
```

Testing: Mocks

Default Answers

- CALLS_REAL_METHODS
- RETURNS_DEEP_STUBS
- RETURNS_DEFAULTS
- RETURNS_MOCKS
- RETURNS_SELF
- RETURNS_SMART_NULLS

Testing: Mocks

Type Safety

```
public interface OngoingStubbing<T> {
  OngoingStubbing<T> thenAnswer(Answer<?> answer);
}
```

Testing: Mocks

Performance

Expensive real implementations replaced by expensive mocks.

- **Runtime code generation**
- **Bytecode manipulation**
- **Reflection 😱**

Testing: Mocks

Performance

```
internal class CoffeeMakerTest {

  private lateinit var heater: Heater

  @Before
  fun setUp() {
    heater = mock {
      on { isHeating } doAnswer { true }
    }
  }
}
```

Testing: Mocks

Dynamic Mutability

```
internal class CoffeeMakerTest {

  private lateinit var heater: Heater

  @Before
  fun setUp() {
    heater = mock {
      on { isHeating } doAnswer { true }
    }
  }

  @Test
  fun `should brew coffee`() {
    // heater already has state!
  }
}
```

Testing: Mocks

Dynamic Mutability

Framework generated mocks introduce a shared, mutable, dynamic, runtime declaration.

Unpredictability

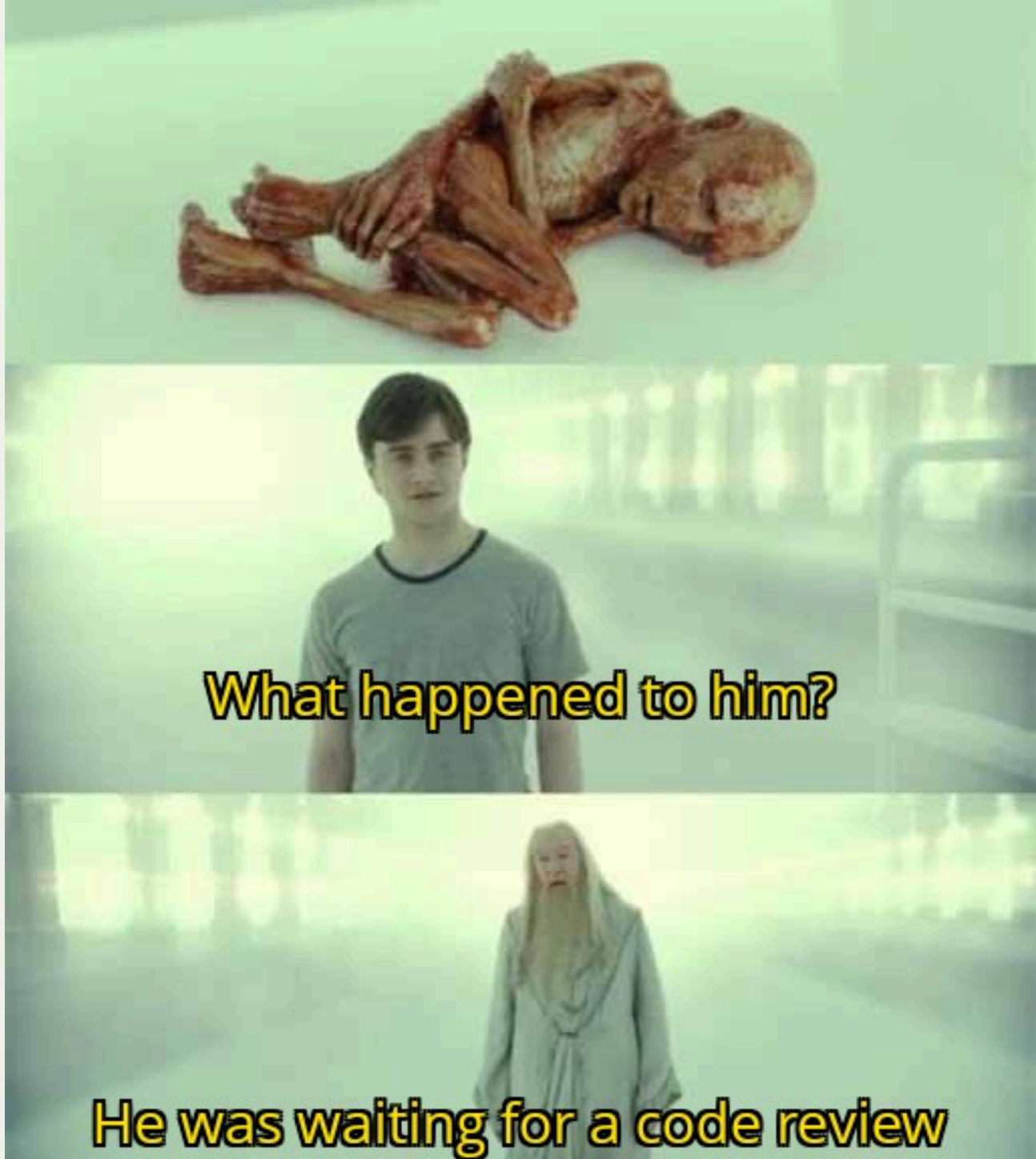
Costs

Unpredictability: Costs

Learning Curve

Unpredictability: Costs

Peer Review



Unpredictability: Costs

Risk of Bugs

Unpredictability: Costs

Slowed Feature Delivery 💸💸💸

Unpredictability

Victims

- Junior developers
- New team members
- Future you

Testing: Mocks

Don't Mock Classes You Don't Own

testing.googleblog.com/2020/07/testing-on-toilet-dont-mock-types-you.html

You Don't Own Your Code!

Your code belongs to your team.

Be considerate.

Testing: Mocks

Considerations

Testing: Mocks

Interaction Verification 👎

Testing: Mocks

What Now?

Testing: Stubs

Testing: Stubs

Simple

```
internal interface Pump {
  fun pump(): Boolean
}

internal object StubPump : Pump {
  override fun pump(): Boolean = true
}
```

Testing: Stubs

Idiomatic

```
internal fun interface Pump {
  fun pump(): Boolean
}

val stub = Pump { true }
```

Testing: Stubs

API Sensitive

```
+ private const val DEFAULT_AMOUNT = 250 // ml
+
- internal fun interface Pump {
+ internal interface Pump {
-   fun pump(): Boolean
+   fun pump(amount: Int = DEFAULT_AMOUNT): Boolean
+ }
```

Testing: Stubs

API Sensitive

```
private const val DEFAULT_AMOUNT = 250 // ml

internal interface Pump {
  fun pump(amount: Int = DEFAULT_AMOUNT): Boolean
}

val stub = Pump { true } // Compilation failure...
```

Testing: Fakes 🏆

Testing: Fakes

```
public class FakePump(private val onPump: (Boolean) -> Boolean) : Pump {

  public val pumped = mutableListOf<Pair<Boolean, Boolean>>()

  override fun pump(full: Boolean): Boolean = onPump(full).also {
    pumped += full to it
  }
}
```

Testing: Fakes

Additional Behaviour

```
private class DelegatingHeater(
  private val delegate: Heater,
) : Heater by delegate {

  private val _drinks = mutableListOf<Any>()
  val drinks: List<Any> by ::_drinks

  override fun <T : Any> heat(body: () -> T): T {
    return delegate.heat(body).also { _drinks += it }
  }
}
```

Testing: Fakes

Authoring

Testing: Fakes

Responsibility

Testing: Fakes

Qualifications

Those who wrote the code are the most uniquely qualified to write the tests.

Testing: Libraries

- androidx.compose.ui:ui-test-junit4
- com.slack.circuit:circuit-test
- io.ktor:ktor-server-test-host
- io.ktor:ktor-client-mock
- kotlinx-coroutines-test

Testing: In Memory

```
internal fun interface CoffeeStore {
  fun has(type: CoffeeType): Boolean
}

internal enum class CoffeeType {
  CAPPUCCINO,
  ESPRESSO,
  LATTE,
}
```

Testing: In Memory

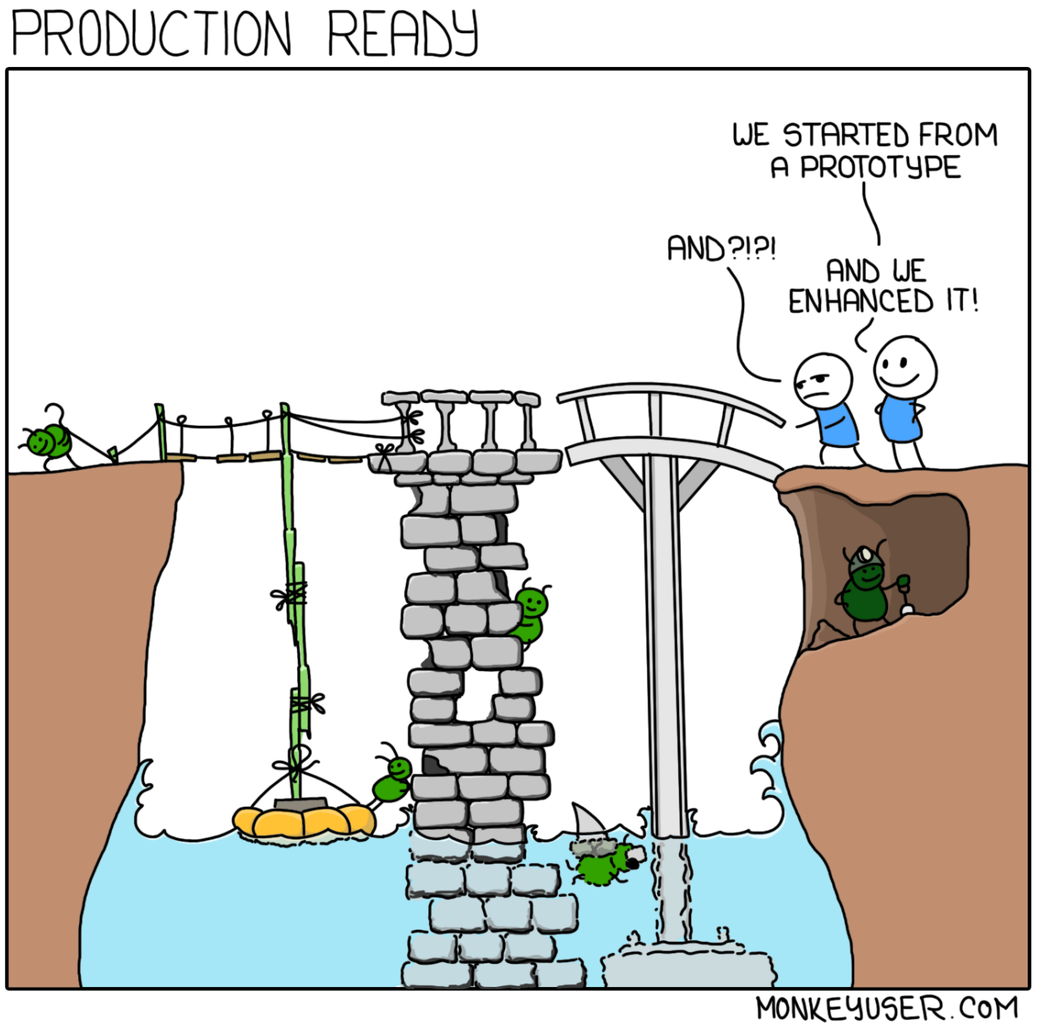
```
internal class InMemoryCoffeeStore : CoffeeStore {

  private val _stock = mutableMapOf<CoffeeType, Int>()
  val stock: Map<CoffeeType, Int> by ::_stock

  override fun has(type: CoffeeType): Boolean {
    return (_stock[type] ?: 0) > 0
  }

  fun add(type: CoffeeType, amount: Int = 1) {
    _stock[type] = (_stock[type] ?: 0) + amount
  }
}
```

Testing: In Memory

Bonus!

In-memory implementations for local or network overrides.

Reality


PRODUCTION READY
WE STARTED FROM A PROTOTYPE
AND?!?!
AND WE ENHANCED IT!
MONKEYUSER.COM

Interface Segregation Principle

No code should be forced to depend on methods it does not use.

Image: dribbble.com/shots/3251806-Interface-Segregation-Principle

Anti-Patterns and Code Smell

Refactoring: Seams 🧵

Refactoring: Seams 🧵

Preprocessing

- **Kotlin Symbol Processing**
- **Kotlin Compiler Plugins**

Refactoring: Seams 🧵

Linking: Classpath

```
import fit.Parser

internal class FitFilter {
  private val parser: Parser =
    Parser.newInstance()
}
```

Refactoring: Seams 🧵

Linking: Classpath

```
buildscript {
    dependencies {
        val googleServicesVersion = libs.versions.google.services.get()
        classpath("com.google.gms:google-services:$googleServicesVersion")
    }
}
```

Refactoring: Seams 🧵

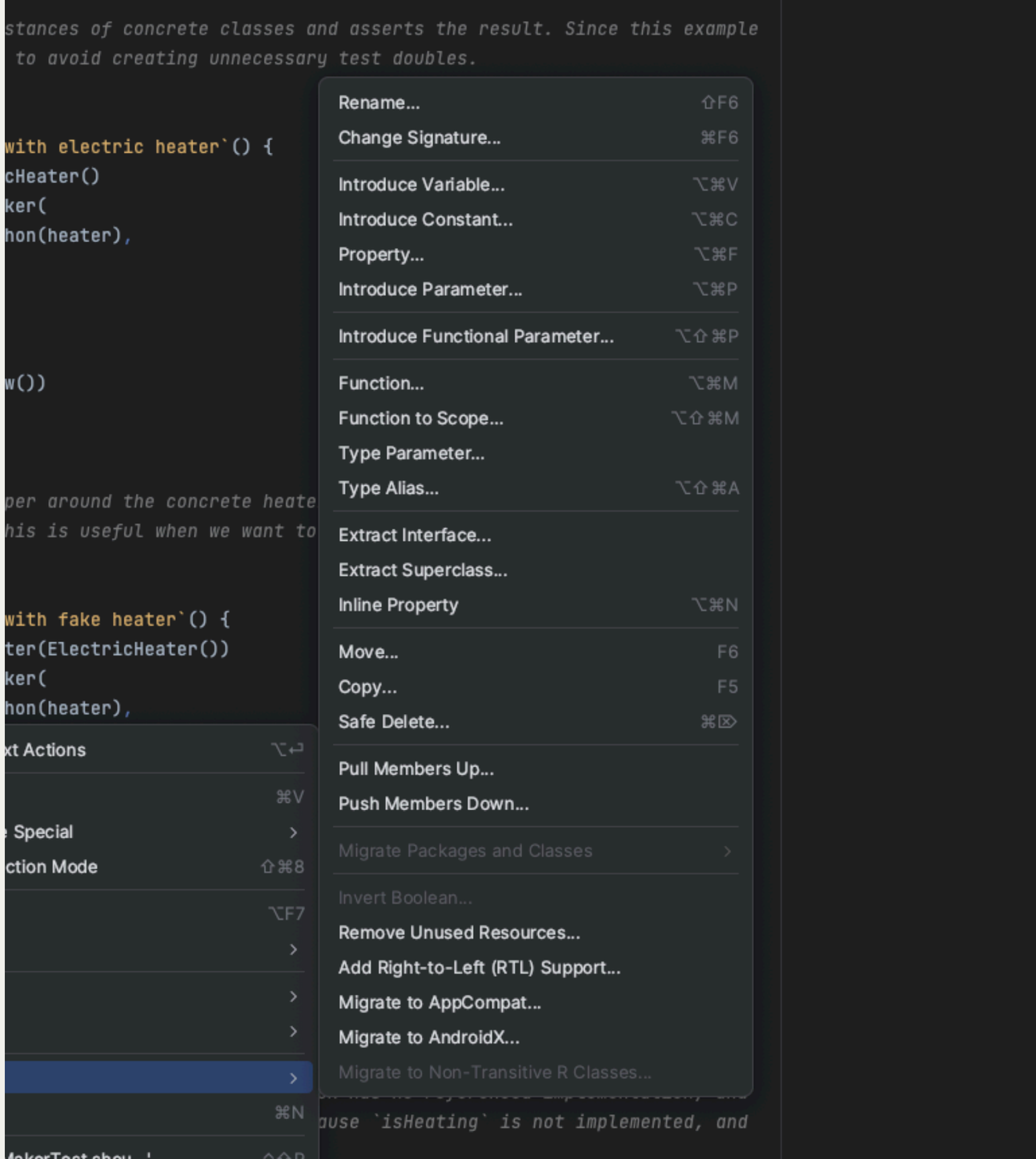
Objects

```
internal class FitFilter {
  private val parser: Parser =
    Parser.newInstance()
}
```

Refactoring: Seams 🧵

Objects: Refactoring

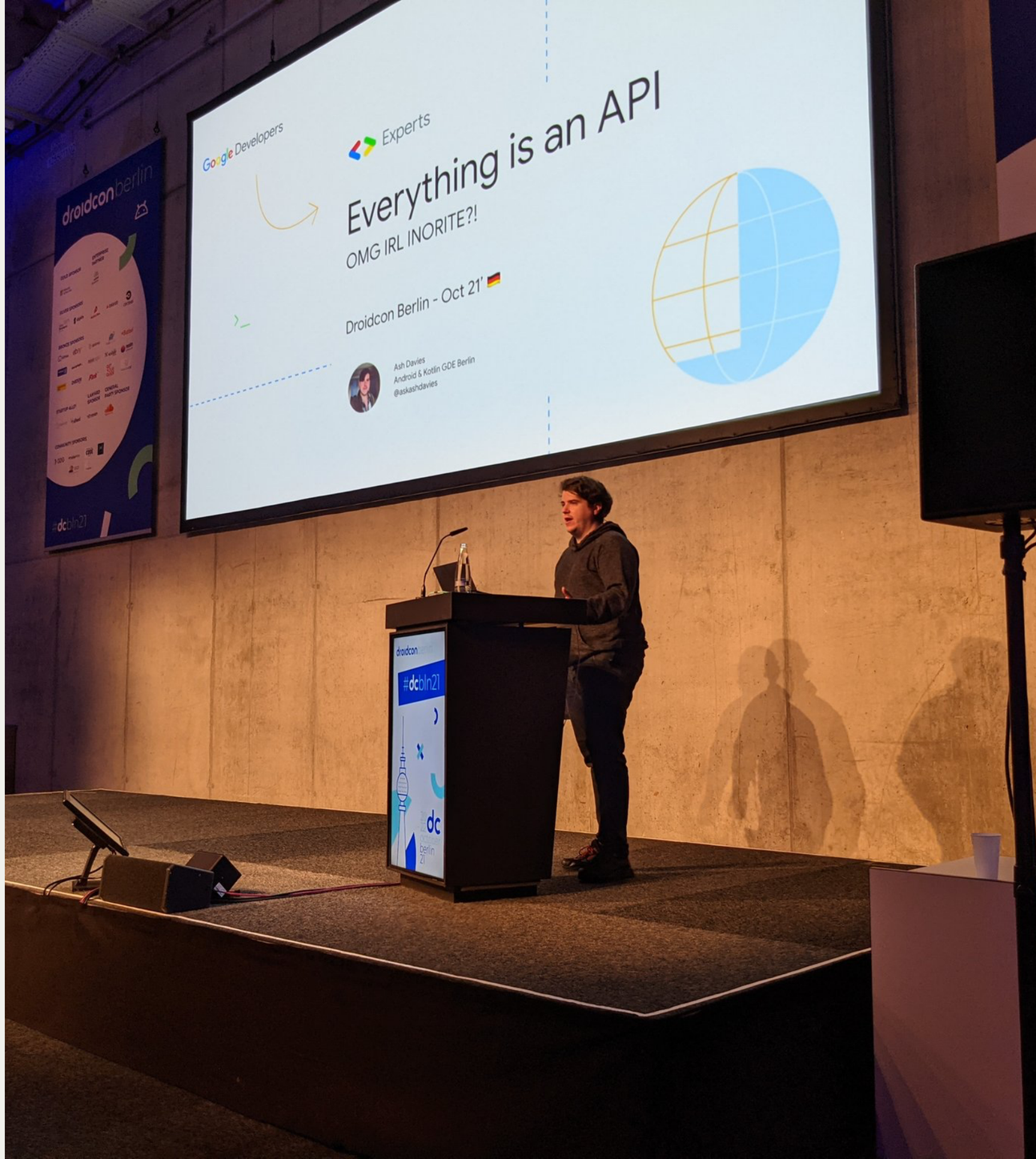
```
================================================================================
diff --git a/FitFilter.kt b/FitFilter.kt

- internal class FitFilter {
-   private val parser: Parser =
-     Parser.newInstance()
- }
-
+ internal fun interface FitFilter {
+   fun filter(input: String): String
+ }
+
+ internal fun FitFilter(parser: Parser) = FitFilter { input ->
+   parser.parse(input)
+ }
```

Everything is an API

ashdavies.dev/talks/everything-is-an-api-berlin-droidcon/

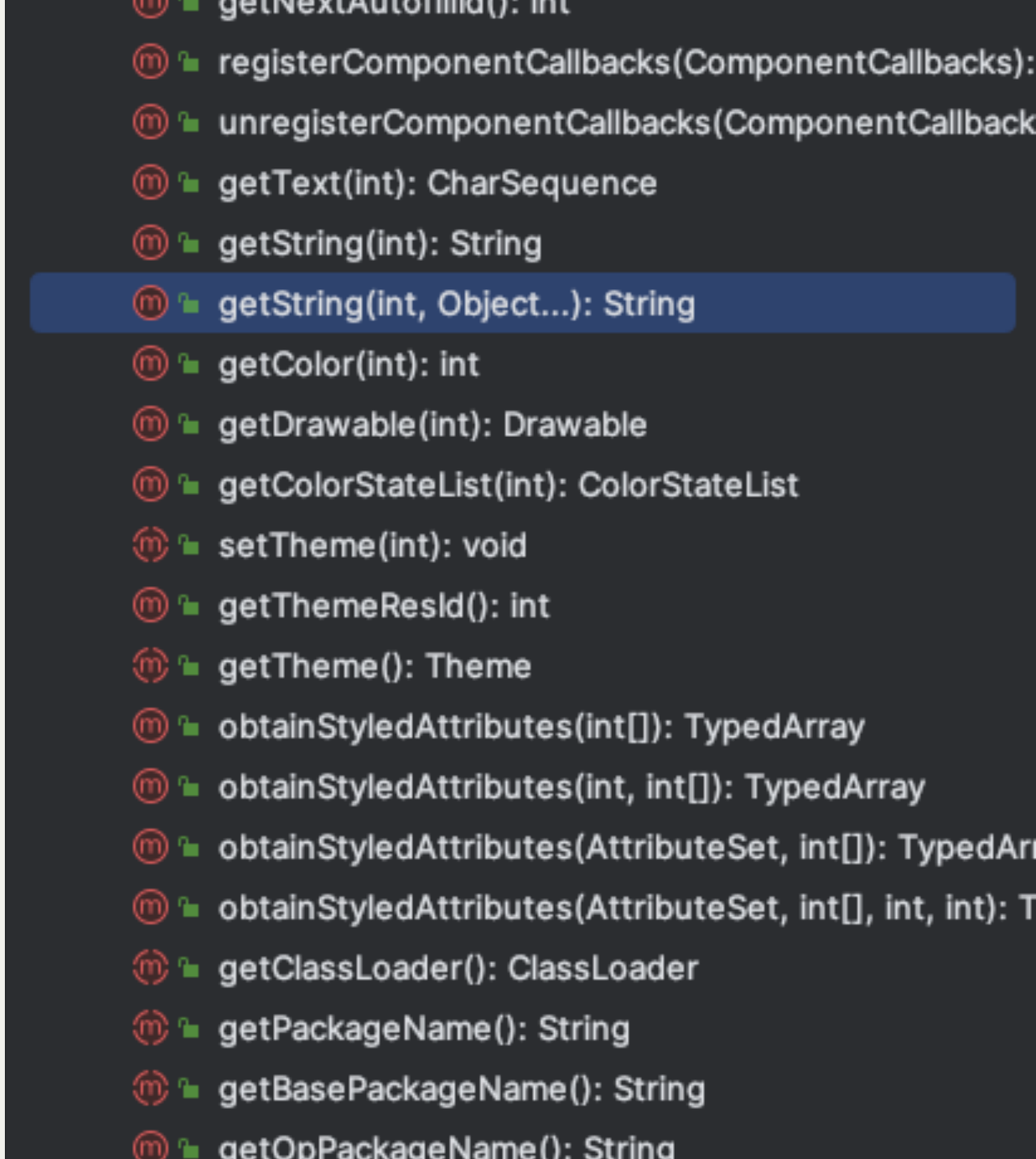
Build versatile and scalable applications with careful API decisions.



Testing: Android

Legacy Inheritance

android.content.Context



God Objects 

Ravioli Code 🥟 💪

Conclusion

- Don’t mock classes you don’t own.
- Don’t mock classes you do own.
- Don’t mock classes (except Context).

Every existing thing is born without reason, prolongs itself out of weakness, and dies by chance.

— Jean-Paul Sartre

Thanks!

github.com/ashdavies/playground.ashdavies.dev/
blob/feature/beyond-the-mockery/
app-launcher/common/src/jvmTest/
kotlin/io/ashdavies/playground/
CoffeeMakerTest.kt

Further Reading

- **Martin Flower: Mocks Aren't Stubs**
  martinfowler.com/articles/mocksArentStubs.html
- **Martin Fowler: Practical Test Pyramid**
  martinfowler.com/articles/practical-test-pyramid.html
- **Images: Monkey User**
  monkeyuser.com
- **Michael Feathers: Working Effectively with Legacy Code**
  ISBN: 978-0-13117-705-5
- **Steve Freeman, Nat Pryce: Growing Object-Oriented Software, Guided by Tests**
  ISBN: 978-0-32150-362-6
- **Testing on the Toilet: Don't mock Types You Don't Own**
  testing.googleblog.com/2020/07/testing-on-toilet-dont-mock-types-you.html
- **Testing on the Toilet: Know Your Test Doubles**
  testing.googleblog.com/2013/07/testing-on-toilet-know-your-test-doubles.html