Testing in Practice

Keeping Your Tests Concise and Declarative

Droidcon Berlin - July '24 🇩🇪

Ash Davies - SumUp

Android / Kotlin GDE Berlin

Cat Person

ashdavies.dev

Why Test?


feat/PRS-519/date-correction-unit-test
Active
1–3 of 3
publication-debug.yml
publication-release.yml
pull-request.yml
sampleapp-pos-android.yml

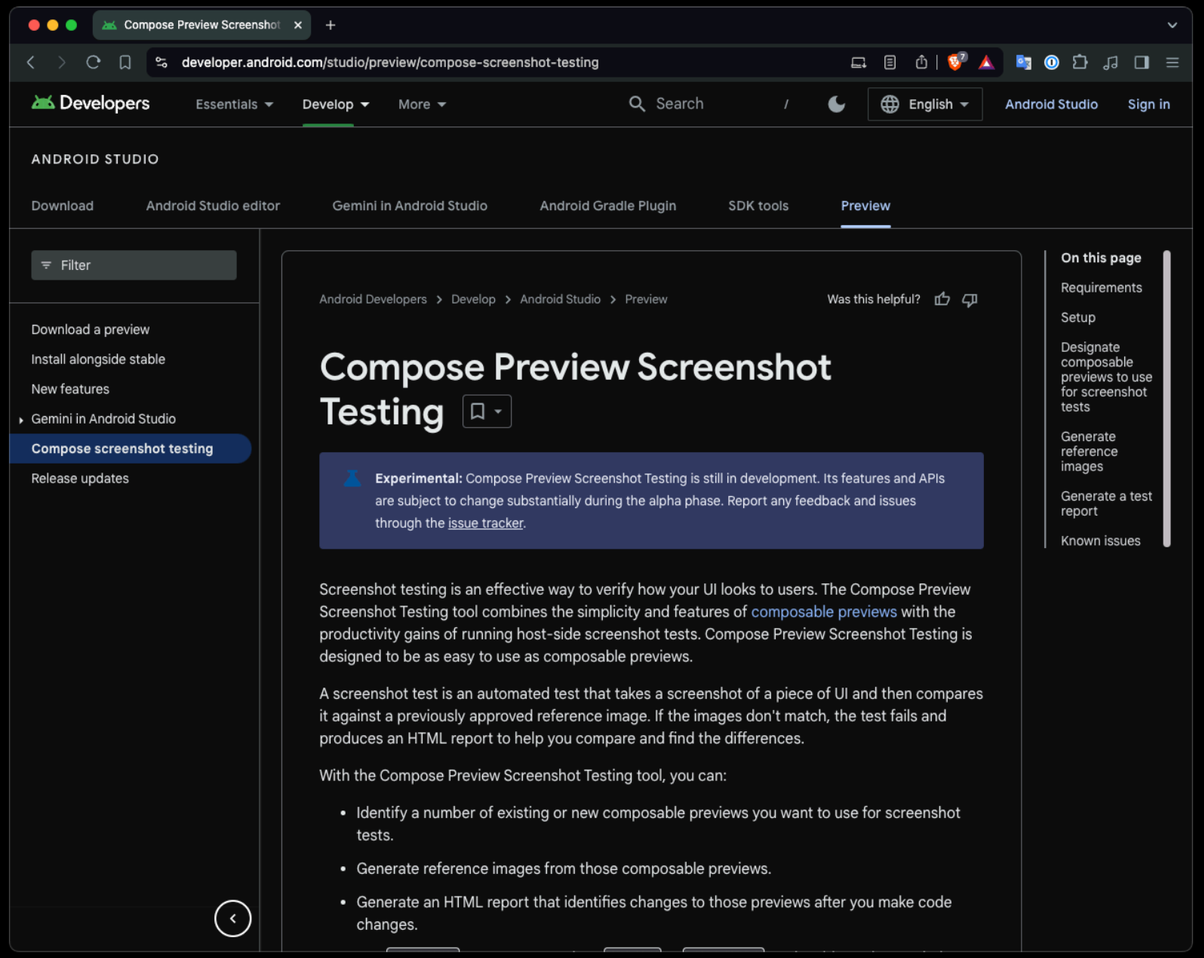
Photos by Daniel Romero, Taylor Vick on Unsplash

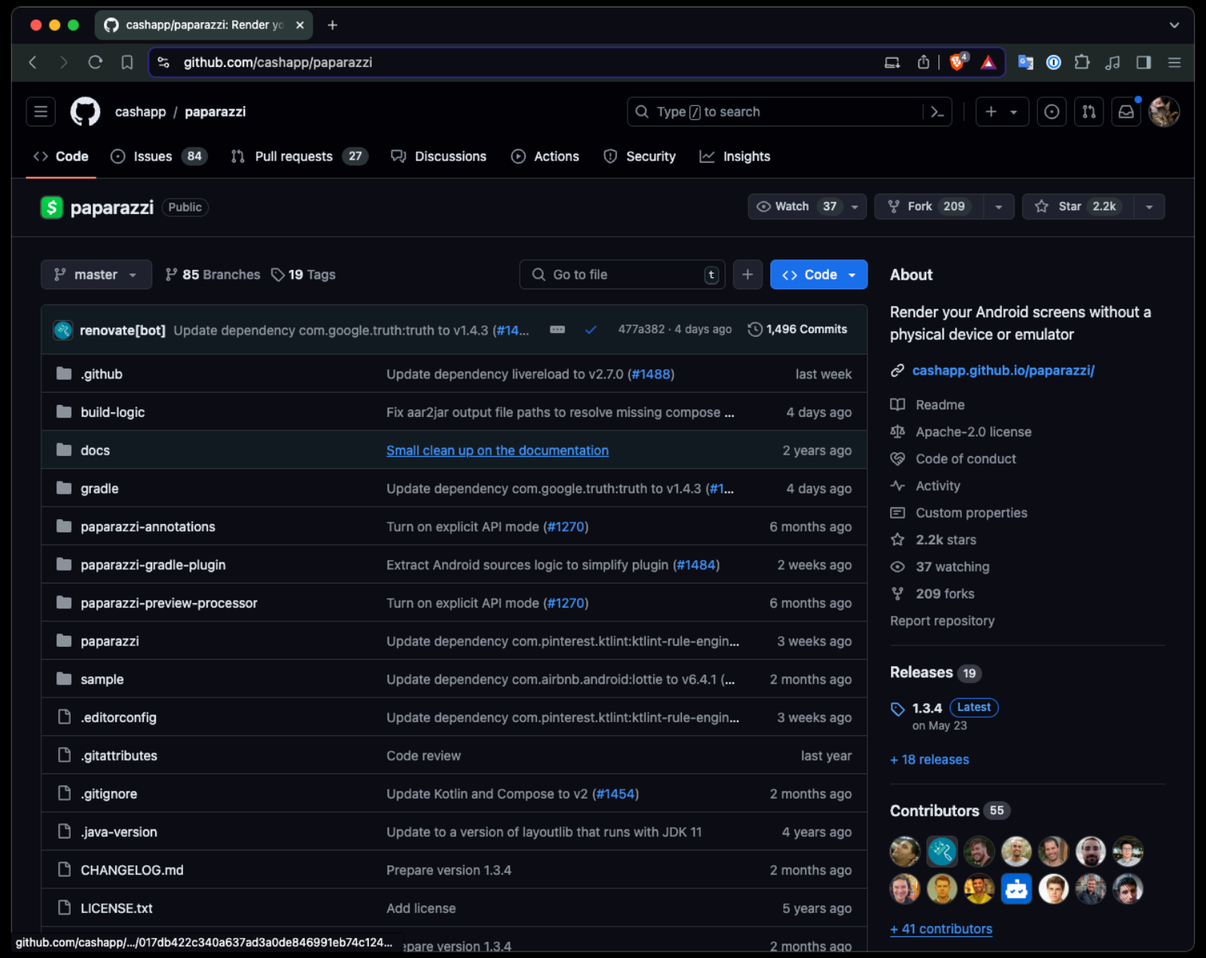
"Hey, I hope you had a relaxing weekend..."

— QA Enginner (8:47 Monday morning)

Testing

- Monkey
- End-to-End
- Acceptance
- Instrumentation
- Integration
- Regression
- Unit


Compose Preview Screenshot
developer.android.com/studio/preview/compose-screenshot-testing
Developers
Essentials
Develop
More
Search
English
Android Studio
Sign in
ANDROID STUDIO
Download
Android Studio editor
Gemini in Android Studio
Android Gradle Plugin
SDK tools
Preview
Filter
Download a preview
Install alongside stable
New features
Gemini in Android Studio
Compose screenshot testing
Release updates
Android Developers › Develop › Android Studio › Preview
Was this helpful?
Compose Preview Screenshot Testing
Experimental: Compose Preview Screenshot Testing is still in development. Its features and APIs are subject to change substantially during the alpha phase. Report any feedback and issues through the issue tracker.
Screenshot testing is an effective way to verify how your UI looks to users. The Compose Preview Screenshot Testing tool combines the simplicity and features of composable previews with the productivity gains of running host-side screenshot tests. Compose Preview Screenshot Testing is designed to be as easy to use as composable previews.
A screenshot test is an automated test that takes a screenshot of a piece of UI and then compares it against a previously approved reference image. If the images don't match, the test fails and produces an HTML report to help you compare and find the differences.
With the Compose Preview Screenshot Testing tool, you can:
Identify a number of existing or new composable previews you want to use for screenshot tests.
Generate reference images from those composable previews.
Generate an HTML report that identifies changes to those previews after you make code changes.
On this page
Requirements
Setup
Designate composable previews to use for screenshot tests
Generate reference images
Generate a test report
Known issues


cashapp/paparazzi: Render yo
github.com/cashapp/paparazzi
cashapp / paparazzi
Type / to search
Code
Issues 84
Pull requests 27
Discussions
Actions
Security
Insights
paparazzi
Public
Watch 37
Fork 209
Star 2.2k
master
85 Branches
19 Tags
Go to file
Code
renovate[bot] Update dependency com.google.truth:truth to v1.4.3 (#14...
477a382 · 4 days ago
1,496 Commits
.github Update dependency livereload to v2.7.0 (#1488) last week
build-logic Fix aar2jar output file paths to resolve missing compose ... 4 days ago
docs Small clean up on the documentation 2 years ago
gradle Update dependency com.google.truth:truth to v1.4.3 (#1... 4 days ago
paparazzi-annotations Turn on explicit API mode (#1270) 6 months ago
paparazzi-gradle-plugin Extract Android sources logic to simplify plugin (#1484) 2 weeks ago
paparazzi-preview-processor Turn on explicit API mode (#1270) 6 months ago
paparazzi Update dependency com.pinterest.ktlint:ktlint-rule-engin... 3 weeks ago
sample Update dependency com.airbnb.android:lottie to v6.4.1 (... 2 months ago
.editorconfig Update dependency com.pinterest.ktlint:ktlint-rule-engin... 3 weeks ago
.gitattributes Code review last year
.gitignore Update Kotlin and Compose to v2 (#1454) 2 months ago
.java-version Update to a version of layoutlib that runs with JDK 11 4 years ago
CHANGELOG.md Prepare version 1.3.4 2 months ago
LICENSE.txt Add license 5 years ago
About
Render your Android screens without a physical device or emulator
cashapp.github.io/paparazzi/
Readme
Apache-2.0 license
Code of conduct
Activity
Custom properties
2.2k stars
37 watching
209 forks
Report repository
Releases 19
1.3.4 Latest
on May 23
+ 18 releases
Contributors 55
+ 41 contributors

Testing

Photo by DiEGO MüLLER on Unsplash

Writing Unit Tests

Pros	Cons
They'll improve the quality of my code	I don't want to
It'll take like 10 mins max	
Literally everyone says that I should	

Writing Unit Tests

Conclusion

I will not write unit tests


60
5
10
15
20
25
30
35
40
45
50
55

"Can we skip the unit tests, just for this feature?"

— PM

Developer: "I'll get back to it later..."

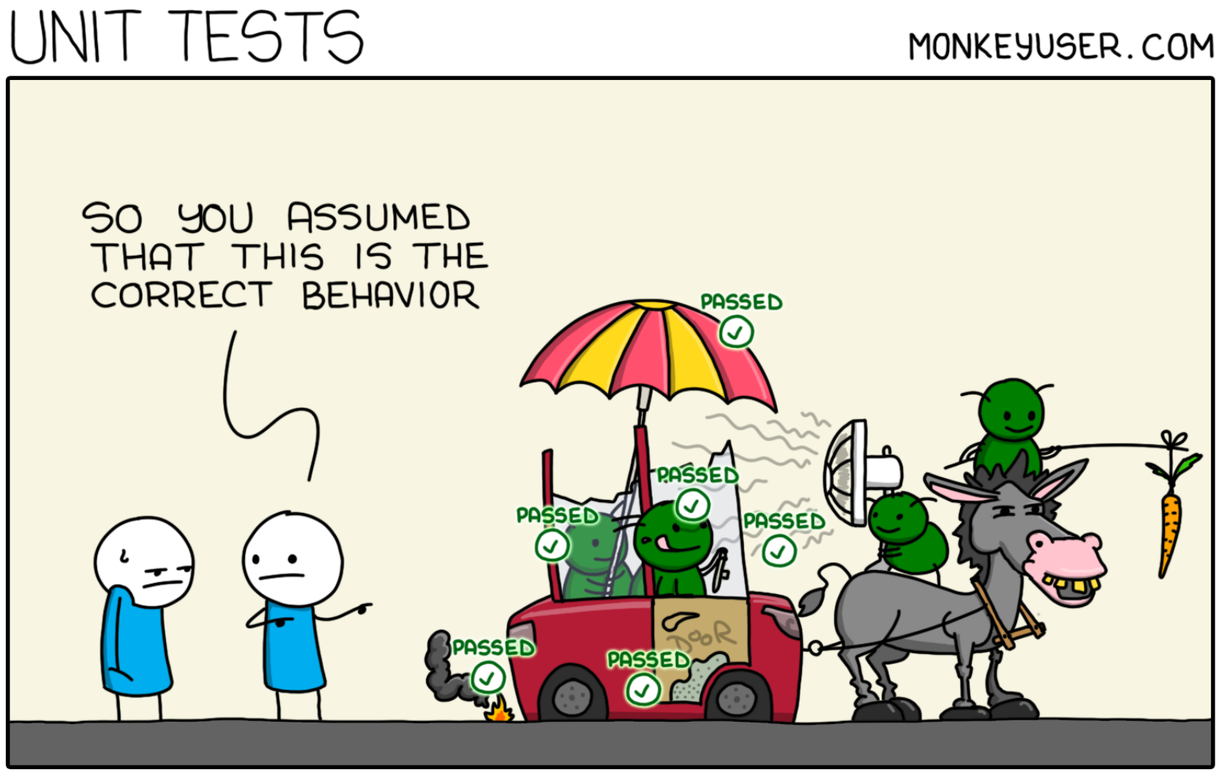
Narrator: "They never got back to it."

"Tests Saved My Ass"

— Ben Kadel

"Our OKR is 100% Code Coverage"

— Former CTO


UNIT TESTS
MONKEYUSER.COM
SO YOU ASSUMED THAT THIS IS THE CORRECT BEHAVIOR
PASSED
PASSED
PASSED
PASSED
PASSED
PASSED
DOOR

Tests test design as well as logic

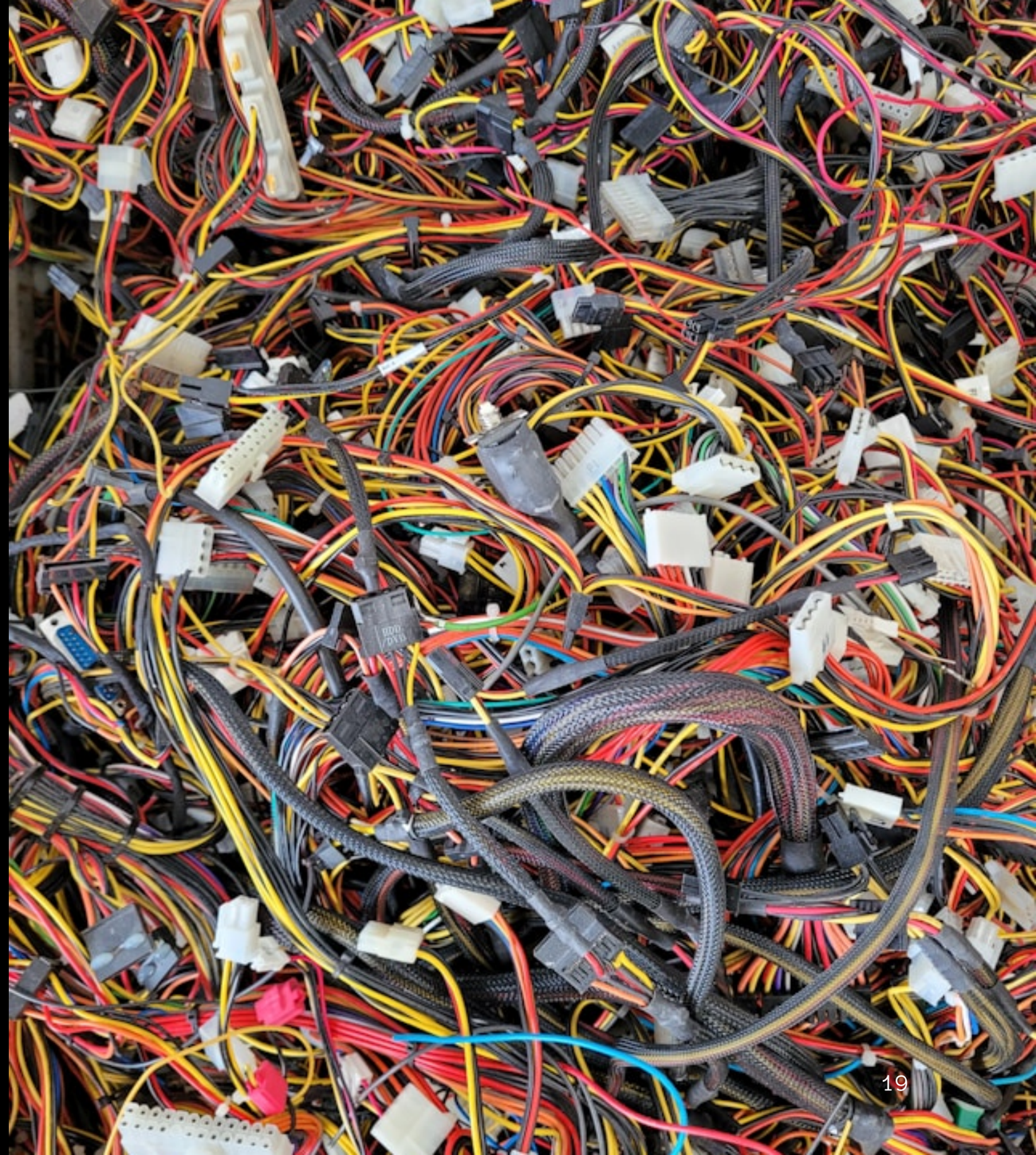
— Michael Feathers

Architecture

Anti-Patterns and Code Smell

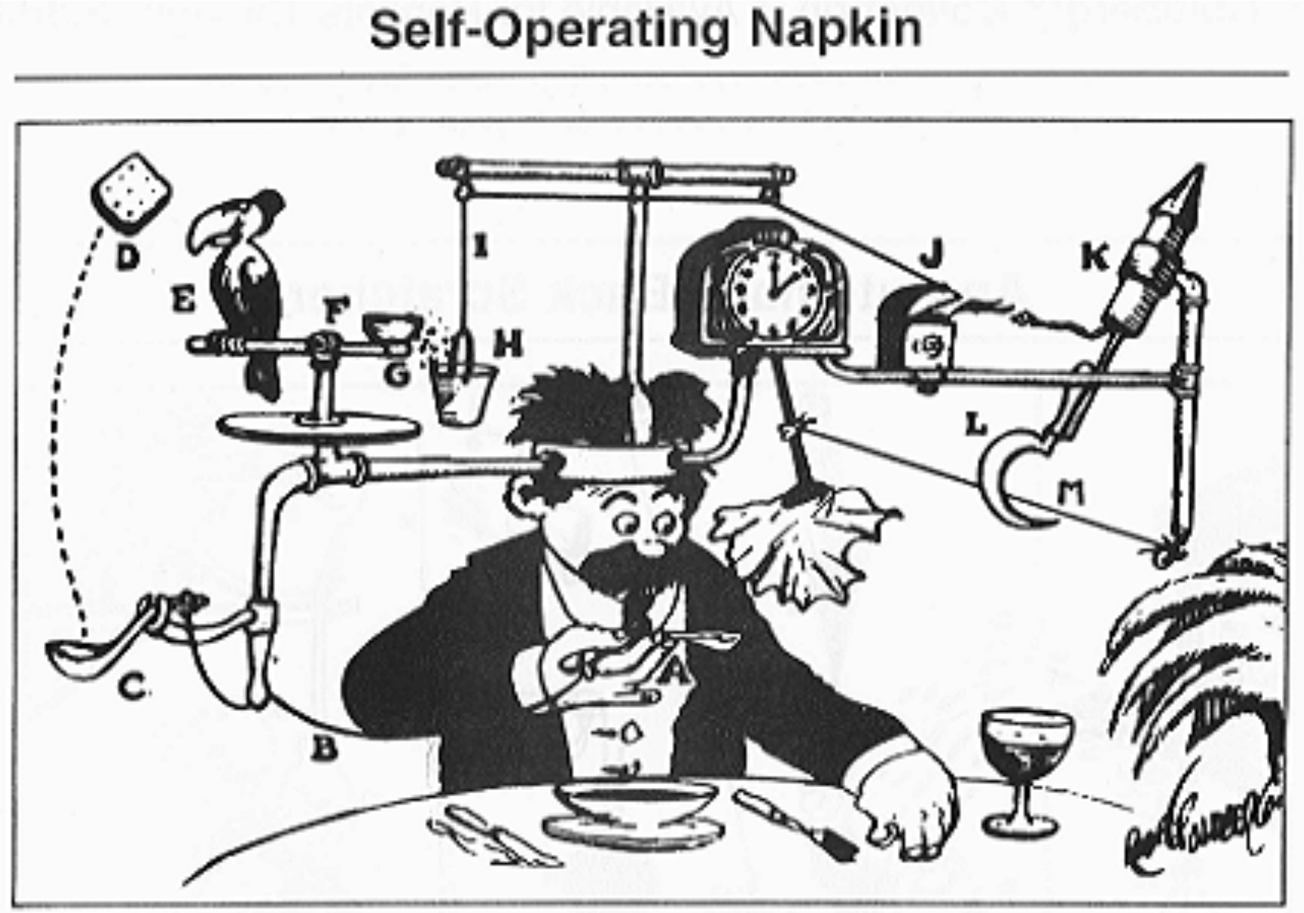
Architecture

- Coupling
- Inheritance
- Mutability
- Polymorphism



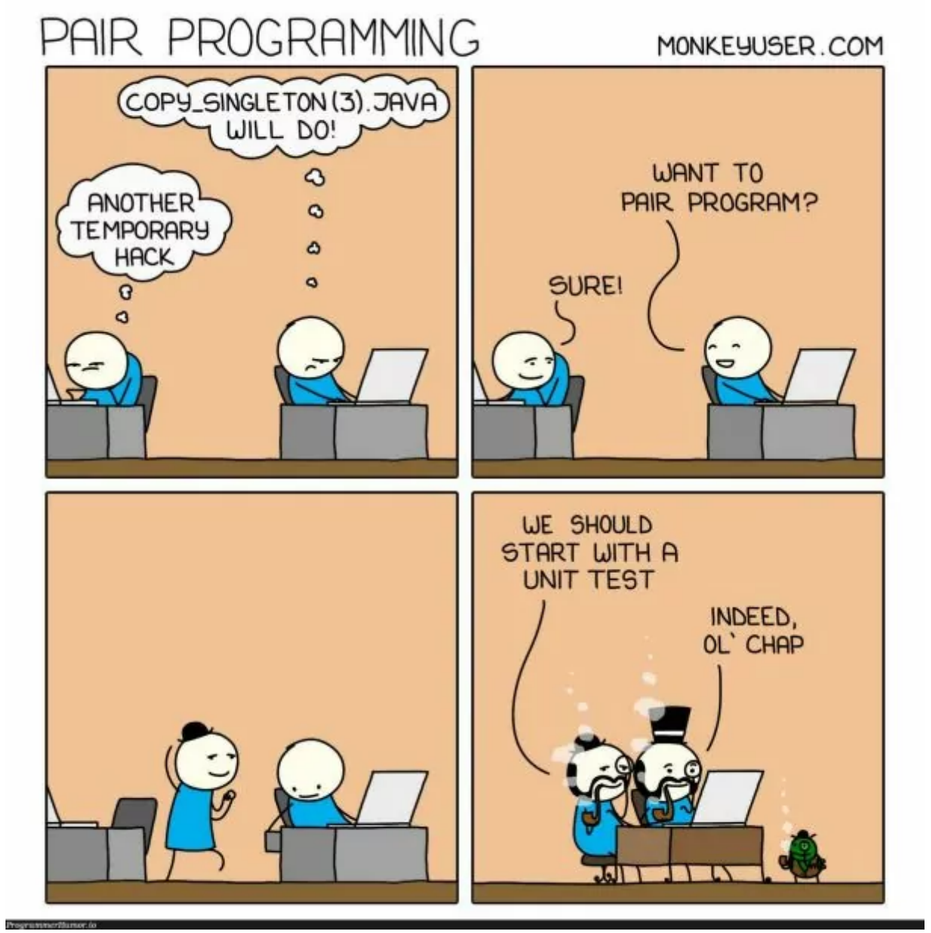
Problem Solving

Solutions Looking for a Problem



Problem Solving

Advent
of Code


PAIR PROGRAMMING
MONKEYUSER.COM
COPY_SINGLETON(3).JAVA WILL DO!
ANOTHER TEMPORARY HACK
WANT TO PAIR PROGRAM?
SURE!
WE SHOULD START WITH A UNIT TEST
INDEED, OL` CHAP

// Let's Code

```
class ThermosiphonTest {

  @Mock
  var heater: Heater

  @Mock
  var logger: Logger

  @InjectMocks
  lateinit var sut: Thermosiphon

  @SetUp
  fun setup() {
    MockAnnotations.init(this)
  }

  @Test
  fun `should heat water`() {
    thermosiphon.pump()

    assertTrue(heater.isHot())
  }
}
```

Test Doubles

Mocks

(ノ°□°)ノ ⁀ ┻━━┻

Test Doubles

Mocks

- Behaviour Verification 👎
- API Insensitivity 😤
- Scale Poorly 📉

```
class ThermosiphonTest {

  @Mock
  var heater: Heater

  @Mock
  var logger: Logger

  @InjectMocks
  lateinit var sut: Thermosiphon

  @SetUp
  fun setup() {
    MockAnnotations.init(this)

    whenever(heater.heat(any())).thenReturn(/* ... */

    whenever(logger.log(any())).thenAnswer {
      /* ... */
  }

  @Test
  fun `should heat water`() {
    thermosiphon.pump()

    val argumentCaptor = argumentCaptor()

    verify(heater).heat(argumentCapture.capture())

    assertTrue(argumentCaptor.values[0])
  }
}
```

Factory Functions

Factory Functions

```
interface Pump {
 fun pump()
}

class Thermosiphon(
  private val heater: Heater,
) : Pump
```

Factory Functions

```
interface Pump {
  fun pump()
}

fun Pump(heater: Heater): Pump {
  return Thermosiphon(heater)
}

private class Thermosiphon(
  private val heater: Heater,
)
```

Factory Functions

CoroutineScope()

CompletableDeferred()

Channel()

MutableStateFlow()

List()

Factory Functions

```
public actual fun <T> lazy(mode: LazyThreadSafetyMode, initializer: () -> T): Lazy<T> =
    when (mode) {
        LazyThreadSafetyMode.SYNCHRONIZED -> SynchronizedLazyImpl(initializer)
        LazyThreadSafetyMode.PUBLICATION -> SafePublicationLazyImpl(initializer)
        LazyThreadSafetyMode.NONE -> UnsafeLazyImpl(initializer)
    }
```

Functional Interfaces

Functional Interfaces

```
fun interface Pump {
  fun pump()
}

fun Pump(heater: Heater): Pump {
  return Thermosiphon(heater)
}

private class Thermosiphon(
  private val heater: Heater,
)
```

Functional Interfaces

```
fun interface Pump {
  fun pump()
}

fun Pump(heater: Heater) = Pump {
  /* ... */
}
```

Functional Interfaces

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

- val pump = mock<Pump> {
-   whenever(heat()).thenReturn(true)
- }

+ val pump = Pump { true }
```

```
class ThermosiphonTest {

  @Test
  fun `should heat water`() {
    var isHot = false

    val heater = Heater { isHot = true }
    val thermosiphon = Thermosiphon(heater)

    thermosiphon.pump()

    assertTrue(isHot)
  }
}
```

```
class ThermosiphonTest {

  @Test
  fun `should heat water`() {
    var isHot = false

    val heater = Heater { isHot = true }
    val thermosiphon = Thermosiphon(heater)

    thermosiphon.pump()

    assertTrue(isHot)
  }

  @Test
  fun `should cool down after heating`() {
    /** ... */
  }
}
```

```kotlin
class ThermosiphonTest {

  @Test
  fun `should heat water`() {
    var isHot = false

    val heater = Heater { isHot = true }
    val thermosiphon = Thermosiphon(heater)

    thermosiphon.pump()

    assertTrue(isHot)
  }

  @Test
  fun `should cool down after heating`() {
    /** ... */
  }

  @Test
  fun `should not catch fire when flying`() {
    /** ... */
  }
}
```

```
class ThermosiphonTest {

  @Test
  fun `should heat water`() {
    var isHot = false

    val heater = Heater { isHot = true }
    val thermosiphon = Thermosiphon(heater)

    thermosiphon.pump()

    assertTrue(isHot)
  }

  @Test
  fun `should cool down after heating`() {
    /** ... */
  }

  @Test
  fun `should not catch fire when flying`() {
    /** ... */
  }

  @Test
  fun `should not become sentient`() {
    /** ... */
  }
}
```

```
class ThermosiphonTest {

  val heater = Heater { isHot = true }
  val thermosiphon = Thermosiphon(heater)

  var isSentient = false
  var hasExploded = false
  var isHot = false

  @Test
  fun `should heat water`() {
    /* ... */
  }

  @Test
  fun `should cool down after heating`() {
    /** ... */
  }

  @Test
  fun `should not catch fire when flying`() {
    /** ... */
  }

  @Test
  fun `should not become sentient`() {
    /** ... */
  }
}
```

```
class ThermosiphonTest {

  val heater = Heater { isHot = true }

  val thermosiphon = Thermosiphon(
    overflow = SmallOverflowTank(),
    heater = heater,
  )

  var isSentient = false
  var hasExploded = false
  var isHot = false

  @Test
  fun `should heat water`() {
    /* ... */
  }

  @Test
  fun `should cool down after heating`() {
    /** ... */
  }

  @Test
  fun `should not catch fire when flying`() {
    /** ... */
  }

  @Test
  fun `should not become sentient`() {
    /** ... */
  }
}
```

```
class ThermosiphonTest {

  val heater = Heater { isHot = true }

  val thermosiphon = Thermosiphon(
    overflow = SmallOverflowTank(),
    aiEngine = GeminiEngine(),
    heater = heater,
  )

  var isSentient = false
  var hasExploded = false
  var isHot = false

  @Test
  fun `should heat water`() {
    /* ... */
  }

  @Test
  fun `should cool down after heating`() {
    /** ... */
  }

  @Test
  fun `should not catch fire when flying`() {
    /** ... */
  }

  @Test
  fun `should not become sentient`() {
    /** ... */
  }
}
```

```
class ThermosiphonTest {

  val heater = Heater { isHot = true }

  val thermosiphon = Thermosiphon(
    overflow = SmallOverflowTank(),
    aiEngine = GeminiEngine(),
    heater = heater,
    launchCodes = emptyList(),
  )

  var isSentient = false
  var hasExploded = false
  var isHot = false

  @Test
  fun `should heat water`() {
    /* ... */
  }

  @Test
  fun `should cool down after heating`() {
    /** ... */
  }

  @Test
  fun `should not catch fire when flying`() {
    /** ... */
  }

  @Test
  fun `should not become sentient`() {
    /** ... */
  }
}
```

```kotlin
class ThermosiphonTest {

  @Test
  fun `should heat water`() {
    var isHot = false
    val thermosiphon = thermosiphon(
      onHeat = { isHot = true },
    )

    thermosiphon.pump()

    assertTrue(isHot)
  }
}

fun thermosiphon(
  onHeat: () -> Unit
) = Thermosiphon(
  overflow = SmallOverflowTank(),
  aiEngine = GeminiEngine(),
  heater = Heater(onHeat),
  launchCodes = emptyList(),
)
```

DRY

Don't Repeat Yourself

- Remove duplication
- High code reusability
- Isolating change

DAMP

Descriptive AND Meaningful Phrases

- Some duplication permitted
- Declarative syntax
- Meaningful naming

DRY vs DAMP

Documentation

Documentation

```
/**
 * Make sure not to change this thing back to the previous implementation,
 * because it breaks on that one specific device in production,
 * when opening the user profile in France using a German locale.
 **/
fun doMyThing() {
  /* ... */
}
```

Documentation

```
/**
 * Make sure not to change this thing back to the previous implementation,
 * because it breaks on that one specific device in production,
 * when opening the user profile in France using a German locale.
 *
 * Update: We changed this to be a remote resolution, this shouldn't happen?
 * We haven't seen it happen in production anymore, but just leave this
 * comment here, incase it occurs again...
 **/
fun doMyThingRefactoredV63() {
  /* ... */
}
```

Documentation

```
/**
 * Make sure not to change this thing back to the previous implementation,
 * because it breaks on that one specific device in production,
 * when opening the user profile in France using a German locale.
 *
 * Update: We changed this to be a remote resolution, this shouldn't happen?
 * We haven't seen it happen in production anymore, but just leave this
 * comment here, incase it occurs again...
 *
 * Update: It's happening again, but for Italian uses in Australia,
 * I really hope our Italian QA enginner doesn't go on vacation to Melbourne again...
 **/
fun doMyThingRefactoredToBeMoreSafeIHopeV91() {
  if (user.locale == Locale.ITALY) {
    /* Hacky hack McHackFace */
  }
}
```

Documentation

Documentation: Tests

```
fun `should store in user specific locale when device is in another country()
```

Documentation: Tests

```
fun `should store in user specific locale when device is in another country() {
  val intendedTarget = ...
  val expectedLocale = ...

  val actualLocale = doMyThing()

  assertEquals(expectedLocale, actualLocal)
}
```

```
commit d4c2d156e78cd579662ac7a658b00ca5aa17fd5d (HEAD -> main, origin/main, origin/HEAD)
Author: Ash Davies <1892070+ashdavies@users.noreply.github.com>
Date:   Sun Jun 23 19:19:44 2024 +0200
```


UNFINISHED WORK
FRIDAY EVENING
TAP
TAP
TAP
PERFECT! I'LL FINISH THIS ON MONDAY
MONDAY MORNING...
WHAT DOES THIS MEAN!?!?
MONKEYUSER.COM

```kotlin
class ThermosiphonTest {

  @Test
  fun `should heat water`() {
    var isHot = false
    val thermosiphon = thermosiphon(
      onHeat = { isHot = true },
    )

    thermosiphon.pump()

    assertTrue(isHot)
  }
}

fun thermosiphon(
  onHeat: () -> Unit
) = Thermosiphon(
  overflow = SmallOverflowTank(),
  aiEngine = GeminiEngine(),
  heater = Heater(onHeat),
  launchCodes = emptyList(),
)
```

What about Context?!

Test Doubles

Mocks

```
fun Context(checkSelfPermission: (String) -> Int): Context = mock {
    whenever(it.checkSelfPermission(any())).thenAnswer { invocation ->
        checkSelfPermission(invocation.arguments[0] as String)
    }
}

val context = Context { PackageManager.PERMISSION_GRANTED }
```

Refactoring

Interface Segregation

Refactoring

Interface Segregation

```
class MenuProvider(
  private val navStateStore: NavStateStore = NavStateStore(),
) {

  fun get() = combine(navStateStore.isEnabled, /* ... */) {
    /* ... */
  }
}

class NavStateStore {

  val isEnabled: Flow<Boolean> = /* ... */

  fun setIsEnabled(value: Boolean) { /* ... */ }

  fun getLastSet(): Long { /* ... */ }
}
```

Refactoring

Interface Segregation

```
class MenuProvider(
  private val navStateStore: NavStateStore = NavStateStore(),
) {

  fun get() = combine(navStateStore.isEnabled, /* ... */) {
    /* ... */
  }
}

class NavStateStore {

  val isEnabled: Flow<Boolean> = /* ... */

  fun setIsEnabled(value: Boolean) { /* ... */ }

  fun getLastSet(): Long { /* ... */ }
}
```

Refactoring

Interface Segregation

```
interface NavStateStore {
  val isEnabled: Flow<Boolean>
}

class InMemoryNavStateStore : NavStateStore {

  override val isEnabled: Flow<Boolean> = /* ... */

  fun setIsEnabled(value: Boolean) { /* ... */ }

  fun getLastSet(): Long { /* ... */ }
}
```

Refactoring

Interface Segregation

```
interface NavStateStore {
  val isEnabled: Flow<Boolean>
}

class PreferencesNavStateStore : NavStateStore {

  override val isEnabled: Flow<Boolean> = /* ... */

  fun setIsEnabled(value: Boolean) { /* ... */ }

  fun getLastSet(): Long { /* ... */ }
}
```

Refactoring

Interface Segregation ✨

```
interface NavStateStore {
  val isEnabled: Flow<Boolean>
}

class SentientNavStateStore : NavStateStore {

  override val isEnabled: Flow<Boolean> = /* ... */

  fun setIsEnabled(value: Boolean) { /* ... */ }

  fun getLastSet(): Long { /* ... */ }
}
```

Refactoring

Interface Segregation

```
interface NavStateStore {
  val isEnabled: Flow<Boolean>
}

class NavStateStoreImpl : NavStateStore {

  override val isEnabled: Flow<Boolean> = /* ... */

  fun setIsEnabled(value: Boolean) { /* ... */ }

  fun getLastSet(): Long { /* ... */ }
}
```

Refactoring

Objects

```
class MenuProvider(
  private val menuDefaults: MenuStateDefaults = MenuStateDefaults,
) {
  /* ... */
}

object MenuStateDefaults {
  val iconWidth = 24
}
```

Refactoring

Objects

```
class MenuProvider(
  private val menuProperties: MenuStateProperties = MenuStateProperties.Default,
) {
  /* ... */
}

interface MenuStateProperties {

  val iconWidth: Int

  companion object Default : MenuStateProperties {
    override val iconWidth = 24
  }
}
```

Refactoring

Objects 🔥

```
class MenuProvider(
  private val menuProperties: MenuStateProperties = MenuStateProperties(),
) {
  /* ... */
}

interface MenuStateProperties {

  val iconWidth: Int

  companion object {

    operator fun invoke(): MenuStateProperties {
      /* ... */
    }
  }
}
```

Refactoring

Factory Function

```
private object MenuStateDefaults {
  val iconWidth = 24
}

class MenuProvider(
  private val menuProperties: MenuStateProperties = MenuStateProperties(),
) {
  /* ... */
}

fun interface MenuStateProperties {
  val iconWidth: Int
}

fun MenuStateProperties(
  iconWidth: Int = MenuStateDefaults.iconWidth,
) = object : MenuStateProperties {
  override val iconWidth = iconWidth
}
```

Testing

Assertions

```
data class WaterState(
  val temperature: Int,
)

fun interface Heater {
  fun heat(water: WaterState): WaterState
}

@Test
fun `should produce water for English Breakfast Tea`() {
  val heater = Heater { it.copy(temperature = 95) }
  val thermosiphon = Thermosiphon(heater)
  val initial = WaterState(21)

  val state = thermosiphon.pump(initial)

  assertTrue(state.temperature >= 95)
}
```

Testing

Assertions

```
 data class WaterState(
   val temperature: Int,
+  val filtered: Boolean,
 )

fun interface Heater {
  fun heat(water: WaterState): WaterState
}

@Test
fun `should produce water for English Breakfast Tea`() {
  val heater = Heater { it.copy(temperature = 95) }
  val thermosiphon = Thermosiphon(heater)
  val initial = WaterState(21, false)

  val state = thermosiphon.pump(initial)

  assertTrue(state.temperature >= 95)
}
```

Testing

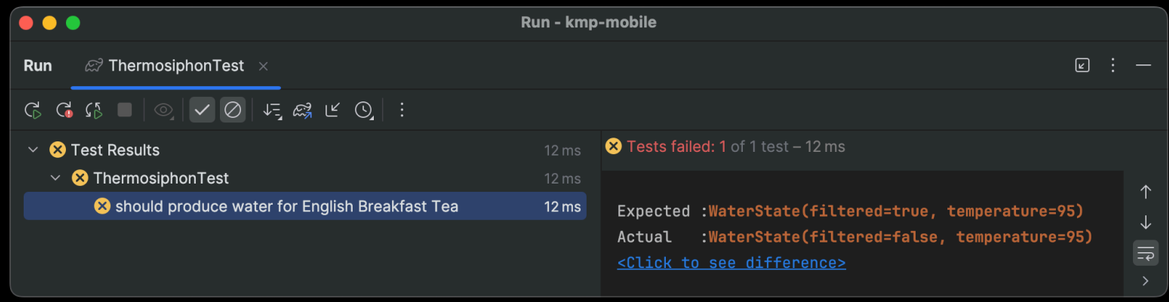
Assertions

```
data class WaterState(
  val filtered: Boolean,
  val temperature: Int,
)

fun interface Heater {
  fun heat(water: WaterState): WaterState
}

@Test
fun `should produce water for English Breakfast Tea`() {
  val heater = Heater { it.copy(temperature = 95) }
  val thermosiphon = Thermosiphon(heater)
  val initial = WaterState(21, true)

  val expected = WaterState(
    filtered = true,
    temperature = 95,
  )

  assertEquals(expected, thermosiphon.pump(initial))
}
```

Testing

Assertions

```
data class WaterState(
  val filtered: Boolean,
  val temperature: Int,
)

fun interface Heater {
  fun heat(water: WaterState): WaterState
}

@Test
fun `should produce water for English Breakfast Tea`() {
  val heater = Heater { it.copy(temperature = 95) }
  val thermosiphon = Thermosiphon(heater)
  val initial = WaterState(21, true)

  val expected = WaterState(
    filtered = true,
    temperature = 95,
  )

  assertEquals(expected, thermosiphon.pump(initial))
}
```



Conclusion

General

- Prefer functional interfaces wherever possible
- Utilise factory functions to isolate behaviour
- Segregate behaviour into smaller interfaces

Conclusion

Testing

- Avoid using mocks unless absolutely necessary
- Restrict test behaviour to function body
- Keep individual tests idempotent
- It's OK to repeat yourself

But wait, there's more!

Out-Takes

Out-Takes: Flaky the Little Flake

android-review.googlesource.com/c/platform/frameworks/support/+/2776638

The story of flaky the little flake

Flaky the little flake was a flaky test that lived happily in the realm of Compose. He loved playing "passing the test suite" with his other test friends. Flaky meant well and tried his best to play the game correctly, but the little flake was a bit awkward and a bit impatient, and often missed. But little flake would always get back up and try again.

His friends often tried to help.

- "Little flake, you are too impatient, you must take your time, wait for the layout and the drawing to be done, so you can assert all your pixels".

But little flake was from the land of Graphics, where color spaces grow strong, where pixels roam free, where eons are measured in milliseconds and eternities in L1 cache misses. So he kept playing with this friends, sometimes making mistakes but always getting back up and trying again.

One day, little flake was playing another game of pass the test suite when a large, scary monster appeared on the playground.

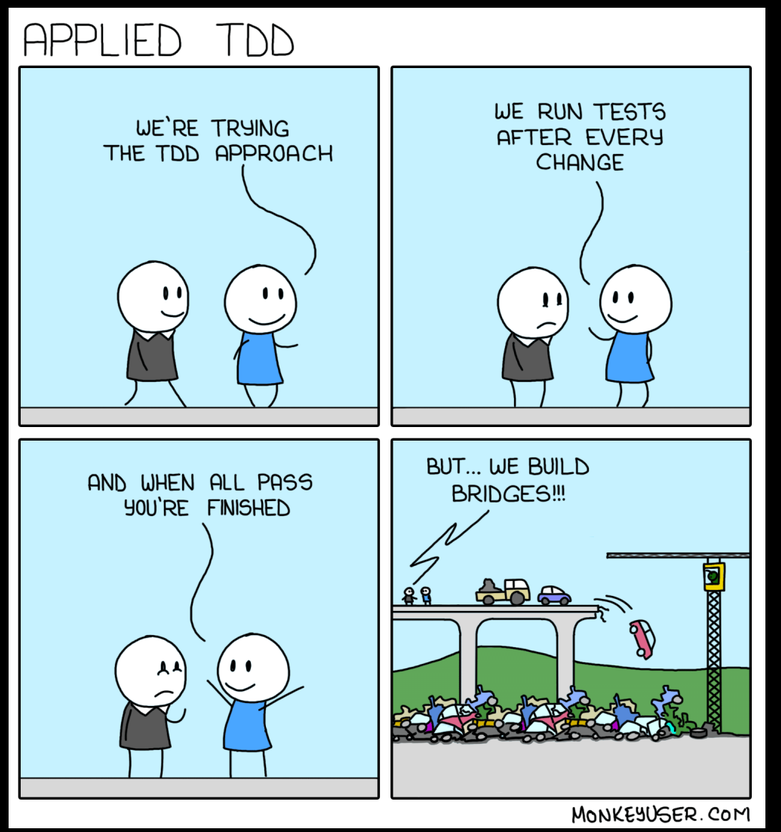
- "Run away little flake! Hide!", said his friends, for the scary monster was a powerful CI Build Bot who had heard about little flake's mistakes.

And so little flake ran away. He was fast and nimble, avoiding detection by the dreaded CI. The monster kept looking and looking, but could not find little flake.

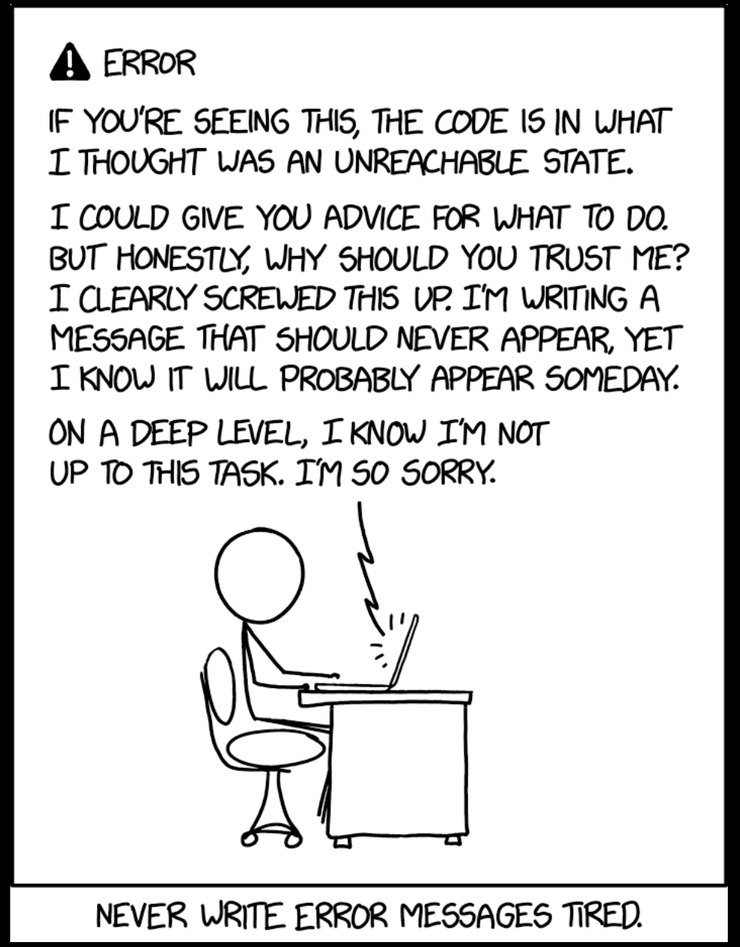
Afraid and all alone, little flake made a decision that would change his life. He decided it was time for him to change his ways and sought out the benevolent Software Engineer to ask for help.

Out-Takes: TDD

```
private fun isEven(number: Int): Boolean {
  // Added to pass unit test
  if (number == 11) {
    return false
  }

  // Added to pass unit test
  if (number == 11) {
    return false
  }

  // Fix for Ticket 12846
  if (number == 11407) {
    return false
  }

  // Fix for Ticket 14336
  if (number == 9) {
    return false
  }

  return true
}
```



Out-Takes: Unreachable State



Thank You!

Ash Davies – SumUp

Android / Kotlin GDE Berlin

ashdavies.dev

Additional

- blog.kotlin-academy.com/item-30-consider-factory-functions-instead-of-constructors-e1c747fc475
- medium.com/@june.pravin/mocking-is-not-practical-use-fakes-e30cc6eaaf4e
- testing.googleblog.com/2024/02/increase-test-fidelity-by-avoiding-mocks.html
- handstandsam.com/2020/06/08/wrapping-mockito-mocks-for-reusability/