TESTING

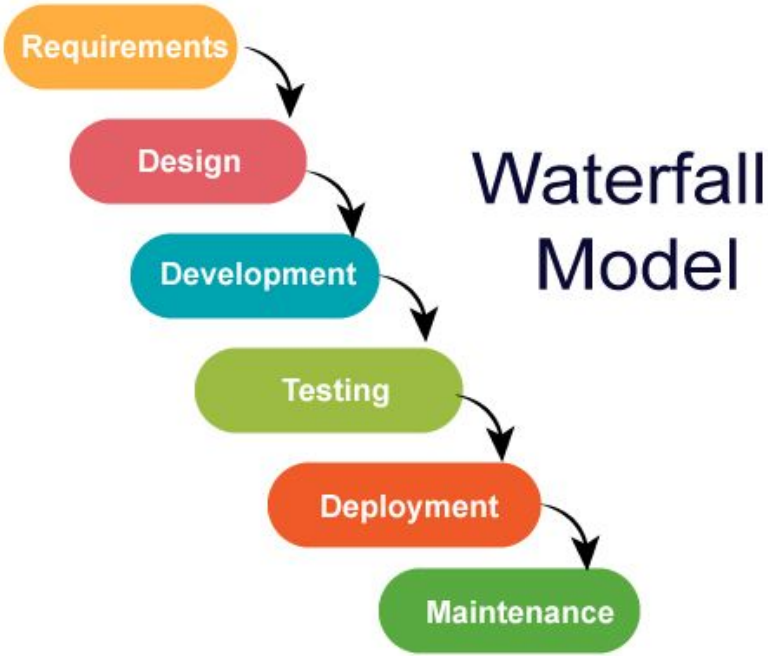
Dominika Gajdova, iOS Engineer at STRV

STRV

INTRODUCTION

01

TESTING


Requirements
Design
Development
Testing
Deployment
Maintenance
Waterfall Model

TESTING

- Software should be
  - Reliable
  - Dependable
  - Secure
  - Available
- Verifies that
  - Requirements have been correctly implemented
  - Software is fit for purpose
  - Defects

“Testing a program demonstrates that it contains errors, never that it is correct.”

Edsger W. Dijkstra

TESTING

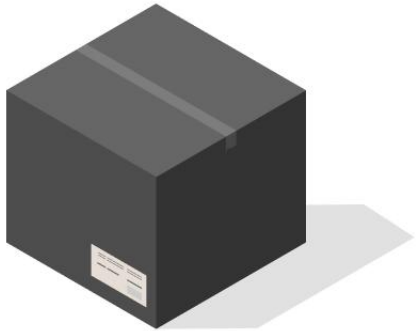
- Building correct and reliable software the first time is very difficult to achieve
- There will always be issues found either by developers or customers
- Iterative process of testing and retesting with a testing group is the key
- Quality assurance role (QA Tester)
- KLOC = one defect per thousand lines of code (metrics)
- Formal mathematical approach (parallel programming)
- Testing social app vs plane controls

TESTING

- Functional – test that app behaves correctly and has the correct functionality
- Non-functional – test operational qualities such as performance, memory or security
- Manual – performed by human tester
- Automated – uses tools and scripts to perform tests
- Regression – re-run existing tests after a change to ensure new code modifications haven't broken previously working features or introduced new bugs
- Smoke – brief main functionality test before deep dive such as opening the app
- Flaky tests — nondeterministic, yield different results

TESTING

QA testers



Black box - we do not know anything

Developers

White box - we know everything

TYPES OF TESTS

02

TYPES OF TESTS

- Unit tests
- Integration tests
- System tests
- Performance tests
- User acceptance tests
- UI tests
- Snapshot tests 📷

XCODE

- Tests are added as new targets
- Test plans – used to manage and run tests under multiple configurations
- Command + U
- Tests can be run from command line as well
- CI (continuous integration) - finding balance and optimizing tests to save costs

UNIT TESTS

- Performed by programmer on completed unit (module)
- Objective is to show that code satisfies design
- Test objective, test procedure, expected result
- *XCTest* (Xcode 5+) and *Swift Testing* (Xcode 16+) frameworks

UNIT TESTS - XCTEST

- Apple testing framework for unit / ui / performance tests
- XCTestCase for defining test cases
- XCTest that can be subclassed
- Assertions based - XCTAssertTrue, XCTAssertEqual etc.
- Tests must start with test prefix – *testTargetExpectation*

UNIT TESTS - XCTEST

```
8  @testable import MyApp
9  import XCTest
10
12 final class MyAppValidationServiceTests: XCTestCase {
12     override func setUpWithError() throws {
13         // Setup before test starts
14     }
15
16     override func tearDownWithError() throws {
17         // Clean up after test finishes
18     }
19
21     func testEmailIsValid() {
21         let sut = ValidationService()
22         let testEmailCorrect = "person@strv.com"
23         XCTAssertTrue(sut.validate(email: email))
24     }
25
27     func testEmailIsInvalid() {
27         let sut = ValidationService()
28         let testEmailIncorrect = "person@strv"
29         XCTAssertFalse(sut.validate(email: email))
30     }
31 }
```

UNIT TESTS - SWIFT TESTING

- Newer Apple testing framework introduced after macros feature
- Simplified and easier to use
- Test prefix no longer needed
- Testing groups
- Parameterized test methods
- *#expect, #require* macros

UNIT TESTS - SWIFT TESTING

```swift
import Testing
@testable import MyApp

struct ValidationServiceTests {
    @Test
    func emailIsValid() async throws {
        let sut = ValidationService()
        let testEmailCorrect = "person@strv.com"
        #expect(sut.validate(email: testEmailCorrect))
    }

    @Test
    func emailIsInvalid() async throws {
        let sut = ValidationService()
        let testEmailIncorrect = "person@strv"
        #expect(!sut.validate(email: testEmailIncorrect))
    }
}
```

UNIT TESTS - SWIFT TESTING - PARAMETERIZED

```swift
enum Food: CaseIterable {
    case burger, iceCream, burrito, noodleBowl, kebab
}


@Test("All foods available", arguments: Food.allCases)
func foodAvailable(_ food: Food) async throws {
    let foodTruck = FoodTruck(selling: food)
    #expect(await foodTruck.cook(food))
}
```

UNIT TESTS - ASYNC CODE

- Async / await is easy to test
- Combine / closure based code is trickier
- Expectations in XCTest with timeouts
- Swift testing – Combine async streams, closure needs to be converted using *withCheckedContinuation*
- Tricky testing values received over time

UI TESTS

- Performed by programmer
- Test user interaction with app’s interface
- Test UI elements and user flows
- XCTest only
- Cooperation with accessibility elements (tagging)

```
struct SomeView: View {
    var body: some View {
        Button("Press me") {
            // action
        }
        .accessibilityLabel("primary-button")
    }
}
```

UI TESTS

- When running UI tests, the entire test can be seen on a simulator
- UI tests are time consuming - parallel tests (but resource heavy)
- Need to be updated each time UI changes
- Enables testing on multiple devices and OS versions

```swift
class SignInViewTests: XCTestCase {
    var app: XCUIApplication!
    var button: XCUIElement {
        app.buttons["signInButton"]
    }

    override func setUpWithError() throws {
        continueAfterFailure = false
        app = XCUIApplication()
        app.launch()
    }

    override func tearDownWithError() throws {
        takeScreenshotOfFailedTest()
        app = nil
    }

    func testEmail() {
        let textField = app.textFields["signInTextField"]
        let myEmail = "xxx.yyy@gmail.com"
        textField.tap()
        textField.typeText(myEmail)
        let typedText = textField.value as? String ?? ""
        XCTAssertEqual(typedText, myEmail)
        XCTAssertEqual(button.isEnabled, true)
    }
```

INTEGRATION TESTS

- Units, components or modules are tested as a combined entity
- Makes sure the components integrate and function together correctly
- Written in same way as unit tests

```
8  import Foundation
9  @testable import SignMeApp
10 import XCTest
11
13 final class SignInStoreTests: XCTestCase {
14     override func setUpWithError() throws {
15         container = DIContainer()
16         signInStore = container.signInStore
17     }
18
18     override func tearDownWithError() throws {
19         container = nil
20         signInStore = nil
21     }
22
24     func testInit() {
25         XCTAssertEqual(signInStore.emailText, "")
26         XCTAssertEqual(signInStore.passwordText, "")
27         XCTAssertEqual(signInStore.buttonDisabled, true)
28     }
28 }
```

PERFORMANCE TESTS

- App launch and loading speed (cold launch ideal < 2s, max 20s)
- CPU and memory usage – avoid battery drain, memory leaks or slowdowns
- Response times – responsiveness of UI interactions
- Stress and load conditions – simulating heavy traffic
- Battery impact
- *XCTestPerformance* API

PERFORMANCE TESTS

```
    func testLaunchPerformance() throws {
        if #available(macOS 10.15, iOS 13.0, tvOS 13.0, watchOS 7.0, *) {
            // This measures how long it takes to launch your application.
            measure(metrics: [XCTApplicationLaunchMetric()]) {   2   Duration (AppLaunch): 0.954 s
                XCUIApplication().launch()
            }
        }
    }
}
```

SNAPSHOT TESTING

- Verifies visual elements by capturing snapshots
- Snapshots are saved usually in the project folder (also published to git repository)
- Each time tests are run, new snapshots are captured and compared with existing ones
- Difference ratio can be adjusted to allow for errors (95% match)
- Safeguarding against accidental UI regressions
- *SnapshotTesting* SPM package from PointFree (external dependency)

SNAPSHOT TESTING

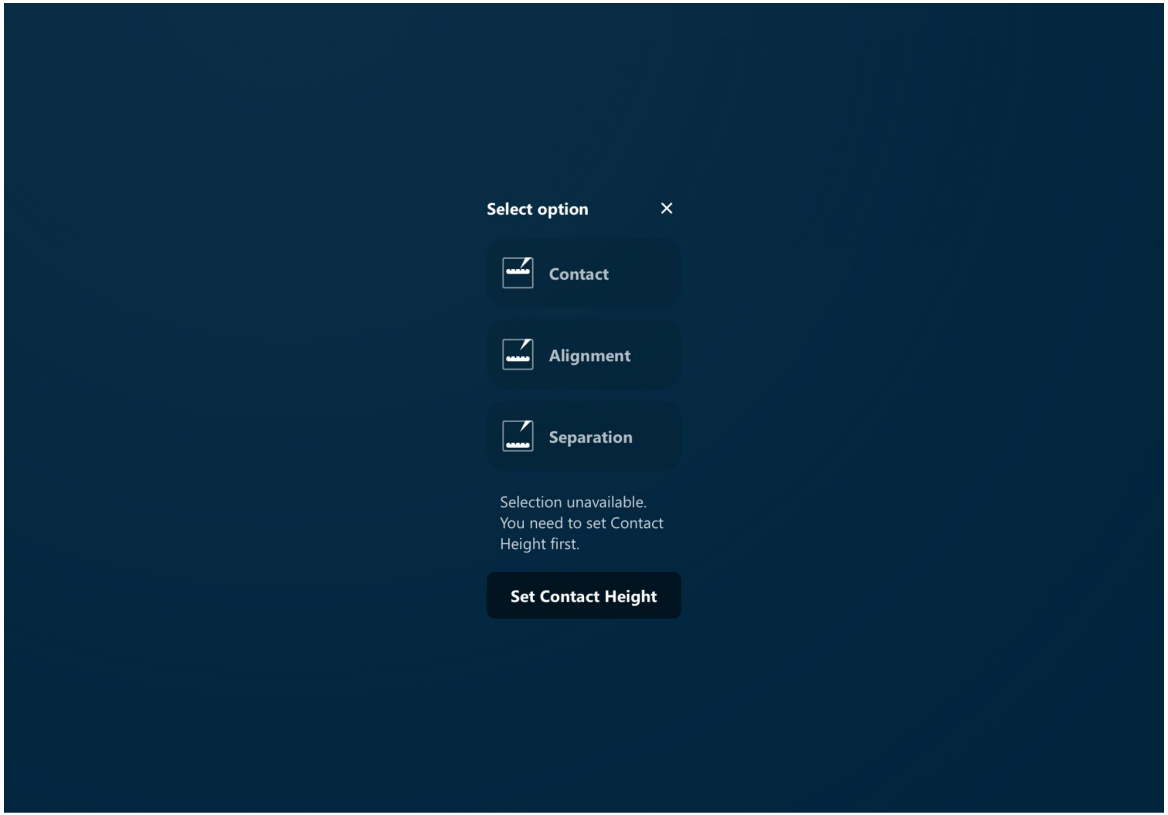
```
final class ChuckContactHeightViewSnapshotTests: SnapshotTests {
    func testContactHeightIsNotSet() {
        let chuckContactHeightView = ChuckContactHeightView(
            store: .init(
                state: .initial,
                eventHandler: self,
                chuckMotionService: ChuckMotionServiceMock(),
                registrationService: RegistrationServiceMock()
            )
        )
        .background(Image(.asset.background))
        .frame(width: 1194, height: 834)

        assertSnapshot(
            matching: chuckContactHeightView,
            as: .image(precision: 0.995)
        )
    }
```

SNAPSHOT TESTING

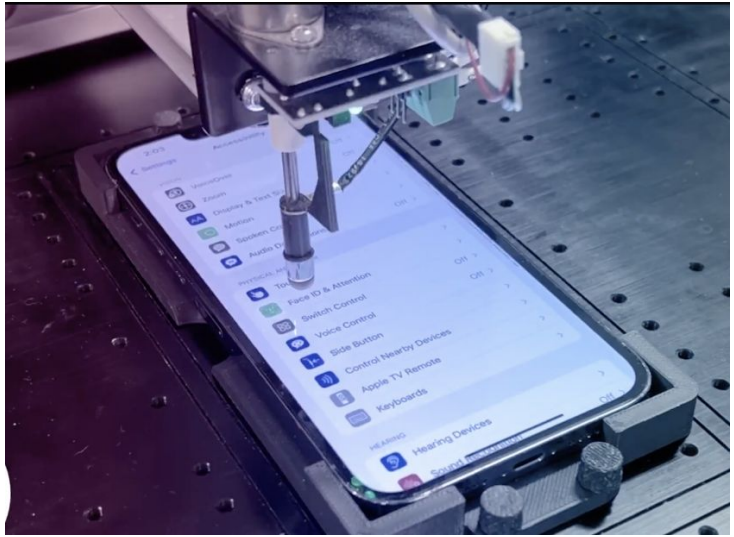
Application	Today at 23:00
__Snapshots__	18. 7. 2025 at 12:39
ChuckContactHeightViewSnapshotTests	13. 8. 2025 at 10:13
testContactHeightAlignmentSelected.1.png	13. 8. 2025 at 10:13
testContactHeightContactSelected.1.png	13. 8. 2025 at 10:13
testContactHeightIsNotSet.1.png	17. 7. 2025 at 10:01
testContactHeightIsSet.1.png	13. 8. 2025 at 10:13
testContactHeightSeparationSelected.1.png	13. 8. 2025 at 10:13
LightAndCameraViewSnapshotTests	16. 10. 2025 at 12:31
testDefault.1.png	17. 7. 2025 at 10:01
testEVue4_10x.1.png	16. 10. 2025 at 12:31
testEVue4_40x.1.png	16. 10. 2025 at 12:31
testLightOn.1.png	17. 7. 2025 at 10:01
testSliders.1.png	17. 7. 2025 at 10:01
LoadWaferViewSnapshotTests	16. 10. 2025 at 8:52
SetupConnectionSnapshotTests	17. 7. 2025 at 10:01
StatusBarSnapshotTests	17. 7. 2025 at 10:01
WaferViewSnapshotTests	17. 7. 2025 at 10:01
ChuckContactHeightViewSnapshotTests.swift	Today at 23:00
Dashboard	13. 8. 2025 at 10:13
__Snapshots__	18. 7. 2025 at 12:37
BlockingStatementViewSnapshotTests.swift	17. 7. 2025 at 10:01
DashboardSnapshotTests.swift	13. 8. 2025 at 10:13
Subviews	13. 8. 2025 at 10:13
__Snapshots__	24. 7. 2025 at 15:34

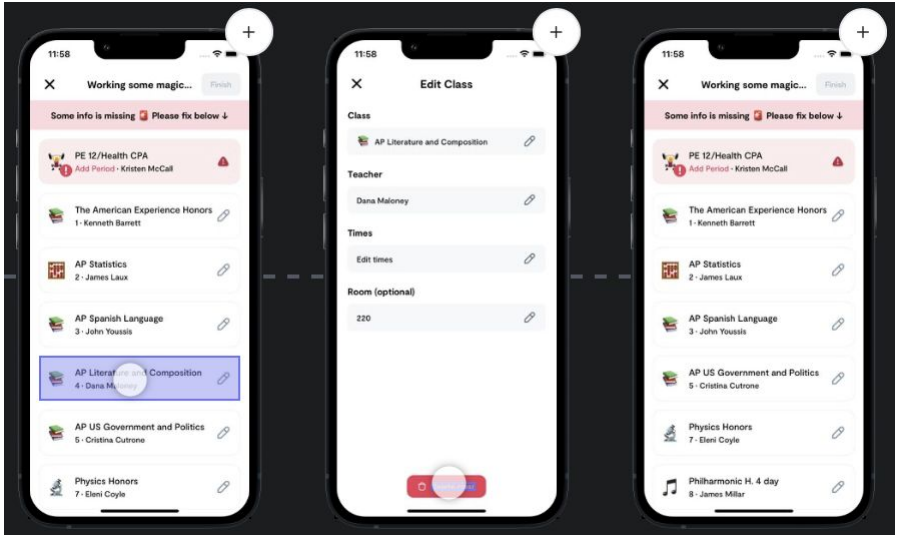
SNAPSHOT TESTING



OTHER TYPES OF TESTS

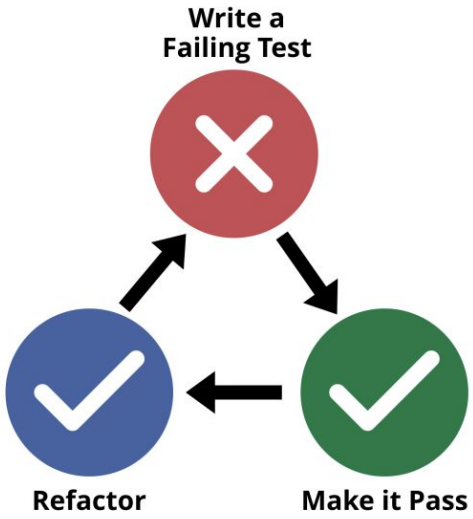
- Benchmark Tests
- 3rd party services (Waldo, Mobot)





TDD

- Test driven development (TDD) - tests are written based on requirements before implementation, then program is coded to satisfy written tests
- Steps
  - 1. Write a failing test
  - 2. Make the test pass
  - 3. Refactor
  - 4. Repeat



CODE COVERAGE

- Percentual value of the amount of lines covered by unit tests
- Statement coverage – measures execution of each line of code
- Branch coverage – ensures every branch in conditional logic (if / else) is tested
- Function coverage – verifies that every method is called
- Path coverage – all possible execution paths
- The higher the code coverage, the better (BUT code coverage says how much it was tested, not how well)
- Fake tests – tests that don’t test anything but add to test coverage

Name	Coverage	Executable Lines
VeloxDash.app	72,2 %	29 070
TemperaturePresets.swift	42,9 %	14
PerformableWithErrorHandling.swift	60,0 %	40
MotionControlButtonWithText.Configuration.swift	100,0 %	23
AutonomousRouter.swift	64,1 %	39
ReadTemperatureChuckStatusResponse.swift	95,5 %	22
JoystickReverse.swift	100,0 %	2
CoordinatorFactory.swift	67,9 %	28
IndexMotionStep.swift	100,0 %	5
Widget1x1View.Configuration+FirstDie.swift	100,0 %	31
ThermalChuckHeatingState.swift	0,0 %	14
ReadProbeStatusResponse.swift	0,0 %	14
SpectrumRouter.swift	72,4 %	29
UIViewController.swift	0,0 %	7
ColorResource+Color.swift	0,0 %	3
ReadPositionRequest.swift	100,0 %	8
EVueService.swift	85,5 %	234
Measurement+FormattedTemperature.swift	100,0 %	10
Widget1x1View.Configuration+ScopeLift.swift	100,0 %	30
EVueRouter.swift	76,5 %	132
ScopeFineFocusButtonsView.swift	91,3 %	104
StageMotionPositionService.swift	88,1 %	67
Double.swift	100,0 %	4
ReadProbeSetupResponse.swift	97,6 %	42

TEST DOUBLES

03

TEST DOUBLES

- Integration tests
- Isolate and focus on the code being tested, not external data sources
- A system under test (**SUT**) may have dependencies on other (complex) objects. We want to eliminate those complex dependencies / objects.
- Example: Instead of calling API to get data, that data is created locally and used during testing.
- Types: *dummy, fake, stub, mock, spy*

```
struct MockAPIService: APIService {
    func fetchData() -> [User] {
        [User(id: 1, name: "Test"), User(id: 2, name: "User")]
    }
}
```

TEST DOUBLES - DUMMY

- Placeholders used to fill parameters but never used during test
- Only purpose is to make the code compile and run

```
let dummyUser = User(id: 0, name: "")
component.process(user: dummyUser) // User object not used
```

TEST DOUBLES - MOCK

- Mocks not only substitute data, they also simulate business logic and record interactions
- Behavioral testing – how functions communicate

```
class MockDatabase: DatabaseProtocol {
    var saveCalled = false
    func save(_ object: Any) { saveCalled = true }
}

mockDatabase.save(data)
XCTAssertTrue(mockDatabase.saveCalled)
```

TEST DOUBLES - STUB

- Provide predefined responses to method calls
- State testing

```
struct StubAPI: WeatherService {
    func fetchTemperature() -> Int { return 25 }
}
```

TEST DOUBLES - FAKE

- Lightweight but functional implementations of real components
- Replicate behaviour using simpler methods (e.g. in-memory database instead of real one)
- Used when more realistic responses are required

TEST DOUBLES - SPY

- Similar to mocks - they also track interactions but instead of working with them they only record them for later inspection

```
class SpyLogger: Logger {
    var messages: [String] = []
    func log(_ message: String) { messages.append(message) }
}
```

MOCKING STORE (VIEWMODEL)

- Provide default values & empty functions
- Useless mock, but needed for SwiftUI Preview

```
// MARK: - Mock
#if DEBUG
extension RelationshipGoalsStore {
    class PreviewStore: RelationshipGoalsStoring {
        var state: RelationshipGoalsState = .initial

        func send(action _: RelationshipGoalsAction) {}
    }
}
#endif
```

MOCKING MODELS

```swift
// MARK: Testing
extension User {
    static let mock: User = User(
        userId: 1,
        id: 1,
        title: "delectus aut autem"
        completed: false
    )
}
```

```swift
// MARK: Testing
extension User {
    static let mockJSONString = """
    {
        "userId": 1,
        "id": 1,
        "title": "delectus aut autem",
        "completed": false
    }
    """
}
```

MOCKING BUSINESS LOGIC

- Faking the whole class
- What if protocol changes?

```
protocol NetworkingProtocol {
    func fetch() -> Int
}

class NetworkManager: NetworkingProtocol {
    func fetch() -> Int {•••}
}

class FakeNetworkManager: NetworkingProtocol {
    func fetch() -> Int {
        return 3
    }
}
```

THANK YOU!

Dominika Gajdova, iOS Engineer at STRV