Unit Testing & TDD Training For iOS & Android Native Apps

Agenda

- Part 1: Unit Testing Concepts
- Part 2: Test-Driven Development
- Part 3: Object Oriented Design for Testing

Unit Testing Concepts

Unit Testing Definition

- A unit test is code made to test a portion of our applications code in an automated manner, verifying methods functionality (result or behavior) of a particular class (SUT or System Under Testing).

Unit Tests Main Properties

- Independent
  - Running a UT should not affect another UT run. Execution order is not important.
- Fast
  - A test which takes several seconds to run is not a Unit Test and probably needs a refactor.
- Isolated
  - UT should not depend on external systems. They need to know their collaborators answers.

Test Class Structure

- Class Setup/TearDown
  - Executed once when the UT class is initialized, not for every test in the class. Useful to avoid test code duplication at class level.
- Test Setup/TearDown
  - Executed once before and after each executed UT. Useful to avoid test code duplication at test level.
- Test Methods
  - The Unit Test implementation itself. Here is where the test run and validations happens.

Android Test Structure Example (Java)

```java
public class PaymentServiceTest {

    private PaymentService paymentService;

    @BeforeClass
    public static void setUpClass() {
        StorageServiceFactory.setStorageService(new InMemoryStorageService());
    }

    @AfterClass
    public static void tearDownClass() { StorageServiceFactory.useDefaultStorageService(); }

    @Before
    public void setUp() {
        StorageService storageService = new InMemoryStorageService();
        StorageServiceFactory.setStorageService(storageService);
        paymentService = new PaymentService(storageService);
    }

    @After
    public void tearDown() { paymentService.dispose(); }

    @Test
    public void comingSoonTest() {
    }
```

iOS Test Structure Example (Swift)

```swift
import XCTest
@testable import FizzBuzz

class BrainTest: XCTestCase {
    let brain = Brain()

    override class func setUp() {
        print("Class setup")
    }

    override class func tearDown() {
        print("Class tear down")
    }

    override func setUp() {
        super.setUp()
        print("test setup")
    }

    override func tearDown() {
        print("test tear down")
        super.tearDown()
    }
```

Tests Naming

- Use **descriptive test names**. It will provide quick feedback when test fails.
- Avoid prefix “test” when no required by framework. Android JUnit does not need it, XCTest require it.
- Choose names which in few words says:
  - **Method** under testing
  - **Condition** under testing
  - **Expected result/behavior**

Tests Naming Examples

- Good examples:
  - Android:

    getPayments_oneRegisteredPayment_returnsListWithRegisteredPayment

  - iOS:

    testListPayments_oneRegisteredPayment_returnsListWithRegisteredPayment

- Bad examples:
  - testCalculateArea2
  - testCalculateArea3

Test Method Organization

- Arrange / Setup
  - Prepare all needed objects/dependencies for test execution.
- Act / Test
  - Test itself it performed, calling one specific method in the system under testing.
- Assert / Verify
  - Method result/behavior verification is performed at the end, using test asserts.

Tests Verification Types

- State Verification
  - Verify the result returned for the method under test, no matter which interactions with other objects has been made. Useful and preferred verification in most cases.

- Behavior Verification
  - Verify that all tested method interactions among other collaboration objects are the expected one (generally done using mocks). Useful for some cases where interactions are important (for example a cache implementation).

Android State Verification Example (Java)

```java
@Test
public void getPayments_oneRegisteredPayment_shouldReturnListWithRegisteredPayment() {
    //Setup
    Payment payment = new Payment(50);
    StorageService storageService = StorageServiceFactory.getStorageService();
    storageService.saveObject(payment);

    //Test
    List<Payment> payments = paymentService.getPayments();

    //Verify
    assertNotNull("getPayments result should not be null.", payments);
    assertEquals("getPayments with invalid amount of payments.", 1, payments.size());
    assertEquals("getPayments did not retrieve previously stored payment.", payment, payments.get(0));
}
```

iOS State Verification Example (Swift)

```swift
func test_getPayments_oneRegisteredPayment_shouldReturnListWithRegisteredPayment() {
    //SetUp
    let payment = Payment(amount: 50)
    let storageService = StorageServiceFactory.getStorageService()
    storageService.saveObject(payment: payment)

    //Test
    let payments: Array<Payment> = paymentService.getPayments()

    //Verify
    XCTAssertNotNil(payments, "getPayments result should not be nil")
    XCTAssertEqual(1, payments.count, "getPayments with invalid amount of payments")
    XCTAssertEqual(payment, payments[0], "getPayments did not retrieve previously stored payment")
}
```

Test Doubles

- Dummy
- Fake
- Stubs
- Mocks

Test Doubles

- Dummy
  - These are objects used to satisfy class dependencies, but they are not used in the test execution itself. For example Java constructor or Swift init dependencies for class instantiation.

- Fake
  - Objects with functional implementation, but with useful shortcuts for testing. For example an in-memory database, which works but after a test run lost all stored data.

Test Doubles

- Stubs
  - Objects with pre-programmed replies. They are not prepared to respond to another thing more than what they were programmed for. Useful in tests for state verifications.

- Mocks
  - Pre-programmed objects with expectations, which are requirements about it use, like: way to be called, invocations amount, invocations parameters. Useful in tests for behavior verifications.

Android Behavior Verification Example (Java - Mockito)

```java
@Test
public void registerPayment_validPaymentData_shouldCallPaymentGatewayWithPaymentData_usingMockito() {
    //Setup - Data
    Payment payment = new Payment(50);
    StorageService mockStorageService = Mockito.mock(StorageService.class);
    PaymentService testPaymentService = new PaymentService(mockStorageService);

    //Test
    testPaymentService.registerPayment(payment);

    //Verify Behavior
    Mockito.verify(mockStorageService, Mockito.times(1)).saveObject(payment);
}
```

Android Behavior Verification Example (Java - EasyMock)

```java
@Test
public void registerPayment_validPaymentData_shouldCallPaymentGatewayWithPaymentData_usingEasyMock() {
    //Setup - Data
    Payment payment = new Payment(50);
    StorageService mockStorageService = EasyMock.createMock(StorageService.class);
    PaymentService testPaymentService = new PaymentService(mockStorageService);

    //Setup - Expectations
    mockStorageService.saveObject(payment);
    EasyMock.expectLastCall();
    EasyMock.replay(mockStorageService);

    //Test
    testPaymentService.registerPayment(payment);

    //Verify
    EasyMock.verify(mockStorageService);
}
```

iOS Behavior Verification Example (Swift)

```swift
func testRegisterPayment_validPaymentData_shouldCallPaymentGatewayWithPaymentData() {
    //Mock
    class MockStorageService : StorageService {

        var saveObjectCount = 0

        override func saveObject(object: NSObject) {
            saveObjectCount += 1
        }
    }

    //Setup
    let payment = Payment(amount: 50)
    let mockStorageService = MockStorageService()
    let testPaymentService = PaymentService(storageService: mockStorageService)

    //Test
    testPaymentService.registerPayment(payment)

    //Verify
    XCTAssertEqual(1, mockStorageService.saveObjectCount, "saveObject was not called.")
}
```

Test-driven Development (TDD)

What TDD is?

- It is a software development process driven by tests.
- It consist in first write one or more unit tests of a class yet to be implemented, then write the minimum necessary code to make that test pass.
- Each Unit Test represents a requirement for that class.
- Each Unit Test keeps and evaluates the developer's knowledge about that class over time.
- It avoids unnecessary code complexity not specified by a particular requirement.

What TDD is not?

- It is not a testing technique.
- It is not write a class first and then write tests to have a good code coverage.
- It is not functional tests, it must be quickly executed and must not have dependencies with external systems.

The TDD process - Step 1

Add a new unit test, that represents a requirement for a method of a class.

The TDD process - Step 2

Execute the test and expect it to fail. In cases when the test don't fail, this means that the requirement has been fulfilled before. Go back to step 1 and add a new unit test for a new requirement.

The TDD process - Step 3

Implement the method in the class, to make the new unit test successfully pass.

The TDD process - Step 4

Execute all the unit test suite, making sure that all the previous test successfully pass. If some tests fail, fix the implementation that make those tests fail. Remember to re execute the test suite every time the code has been changed.

The TDD process - Step 5

Improvements and cleaning code, this can be made without worries about breaking something because the test suite gives us immediate feedback if some test fails

TDD Process in a nutshell

1. Add a new test
2. Execute the test
3. Add the least amount of code necessary to make that test pass
4. Execute the whole test suite
5. Refactor and clean code

Tips

- Any development can be faced using TDD starting with an interface with the signature of the methods to be implemented and tested.
- One single person can be in charge to implement the tests and other person can be in charge to implement the class methods.
- An alternative to avoid the interface is to define the class with its method by simply throwing an exception indicating that the method has not yet been implemented.

Object Oriented Design for Testing

Dependency Injection

- Allows that dependent objects needed in our class implementation can be **specified (injected) from outside** of that class.
- This can be done using constructors/initializers or properties.

Dependency Injection - Using Constructor (Java)

```java
public class MyClassWithConstructorDI {

    private final PaymentService paymentService;

    public MyClassWithConstructorDI(PaymentService paymentService) {
        if (paymentService == null) {
            throw new IllegalArgumentException("Argument paymentService null.");
        }

        this.paymentService = paymentService;
    }

    public void verifyAndRegisterPayment(Payment payment) {
        if (payment != null) {
            paymentService.registerPayment(payment);
        }
    }
}
```

Dependency Injection - Using Initializer (Swift)

```swift
import UIKit

class MyClassWithConstructorDI: NSObject {
    private var paymentService: PaymentService? = nil

    convenience init(paymentService: PaymentService) {
        self.paymentService = paymentService
        self.init()
    }

    func verifyAndRegisterPayment(payment: Payment) {
        paymentService?.registerPayment(payment: payment)
    }
}
```

Dependency Injection - Using Setter (Java)

```java
public class MyClassWithSetterDI {

    private PaymentService paymentService;

    public void setPaymentService(PaymentService paymentService) {
        this.paymentService = paymentService;
    }

    public void verifyAndRegisterPayment(Payment payment) {
        verifyPaymentServiceIsInitialized();

        if (payment != null) {
            paymentService.registerPayment(payment);
        }
    }

    private void verifyPaymentServiceIsInitialized() {
        if (paymentService == null) {
            throw new IllegalStateException("paymentService not yet initialized.");
        }
    }
}
```

Dependency Injection - Using Property (Swift)

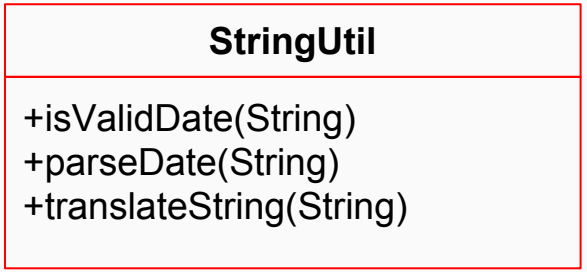
```swift
import UIKit

class MyClassWithSetterDI: NSObject {
    private var paymentService: PaymentService? = nil

    func setPaymentService(paymentService: PaymentService) {
        self.paymentService = paymentService
    }

    func verifyAndRegisterPayment(payment: Payment) {
        paymentService?.registerPayment(payment: payment)
    }
}
```

Single Responsibility

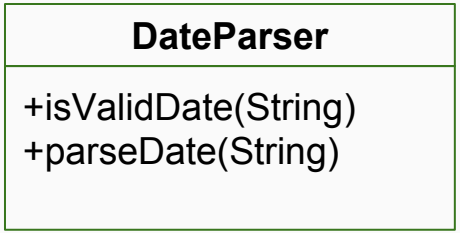
- Each class should have a single responsibility.
- **Reduces coupling** of different domain functionalities.
- Facilitates classes **reusage**.
- The opposite of having Utils/Helper classes. These classes hardly have a unique responsibility.

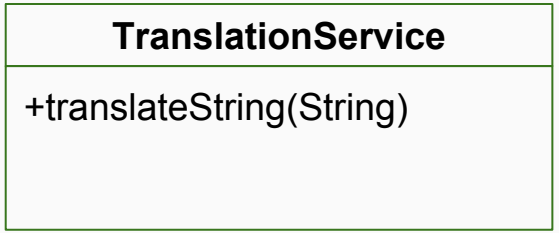
Single Responsibility - Examples

- Bad Example:



- Good Examples:





Don’t talk to strangers (1)

- An object only should **invoke methods of direct known dependencies**.
- Avoid calling dependencies of our dependencies (strangers). This is avoid chain of methods in code.

Don’t talk to strangers (2)

- It’s a guide, not a rule. For example this could be avoid in model/entities objects.
- **Reduces object coupling**, improving code maintainability and allows refactor when needed.

Don’t talk to strangers - Examples

- Bad Examples:

  Java: this.paymentService.getStorageService().getStoredObject(paymentId)

  Swift: self.paymentService.storageService.storedObjectWithId(paymentId)

- Good Examples:

  Java: this.paymentService.getPayment(payement)

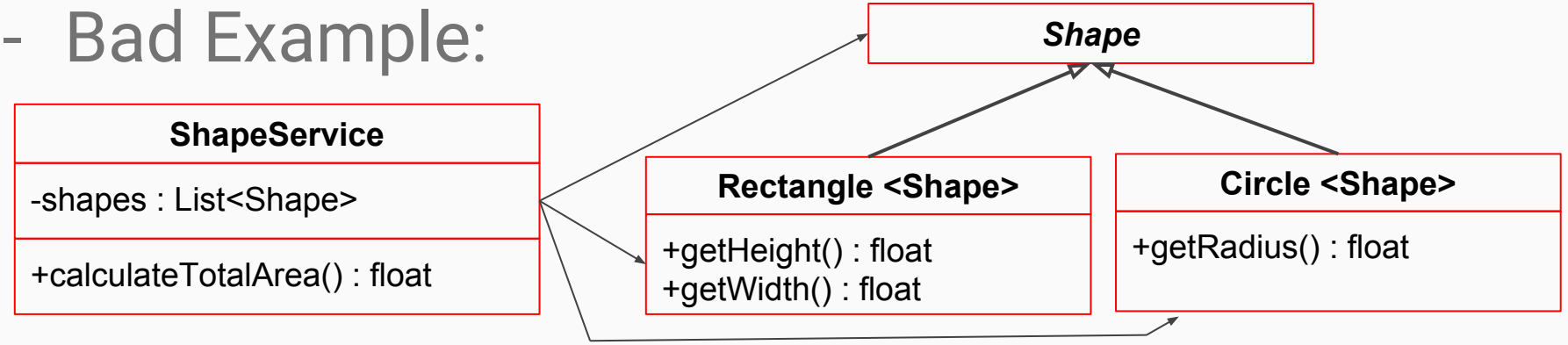
  Swift: self.paymentService.paymentWithId(paymentId)

Tell don’t ask

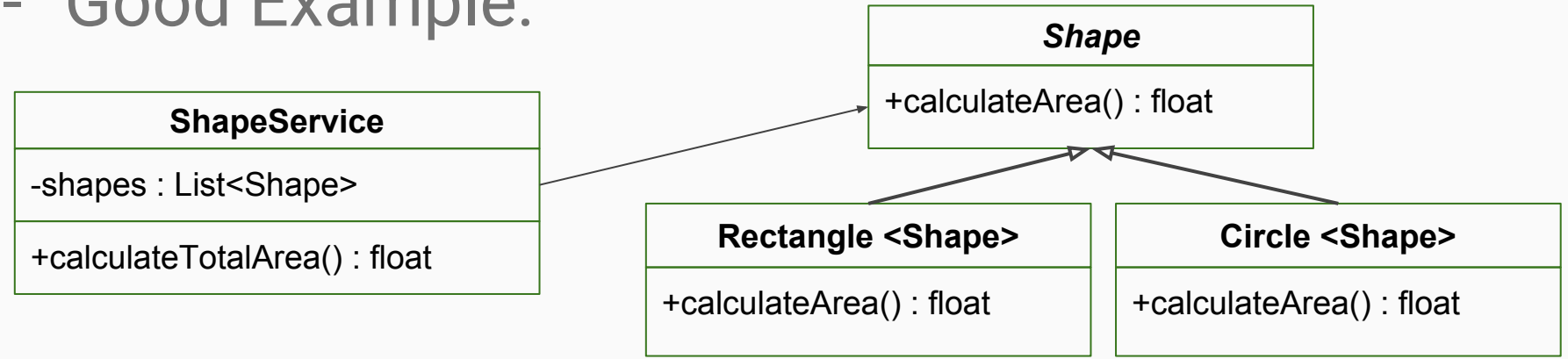
- Objects should **tell another objects to perform actions** instead of asking for data and processing the data itself.
- Objects should be as **lazy** as possible and **delegate** the processing to its dependencies.
- Improves class **testability** and class **cohesion**.

Tell don’t ask - Examples

- Bad Example:



- Good Example:

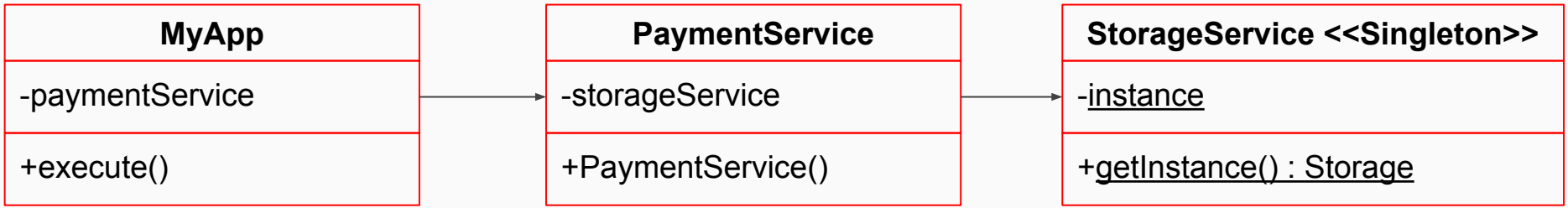


Avoid Singletons

- Singleton is a pattern that share a **global state** in the whole application.
- In unit testing, does not allow dependencies control, because it **hides class dependencies** to the consumer.
- Test execution order can affect the result of the test run when using singletons.

Avoid Singletons - Examples

- Bad Example:



- Good Example:

Questions?

Thank you so much!