Testing

Week 7

If you're not failing, you're not trying hard enough.

— Martin Fowler

Objectives

- Understand the importance of software testing
- Understand various types of testing

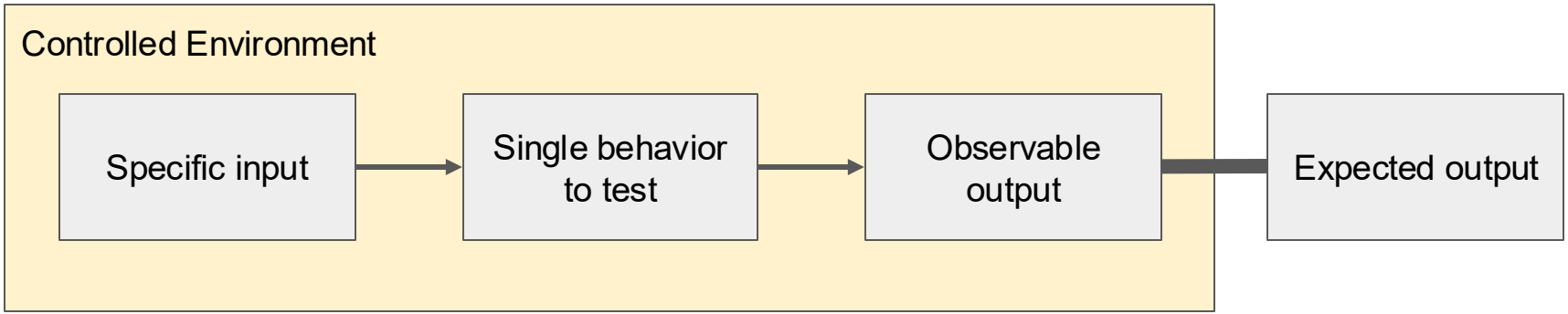
Content

- Definition and importance of testing
- Types of testing
- Testing for our project
  - Unit testing
  - Test doubles

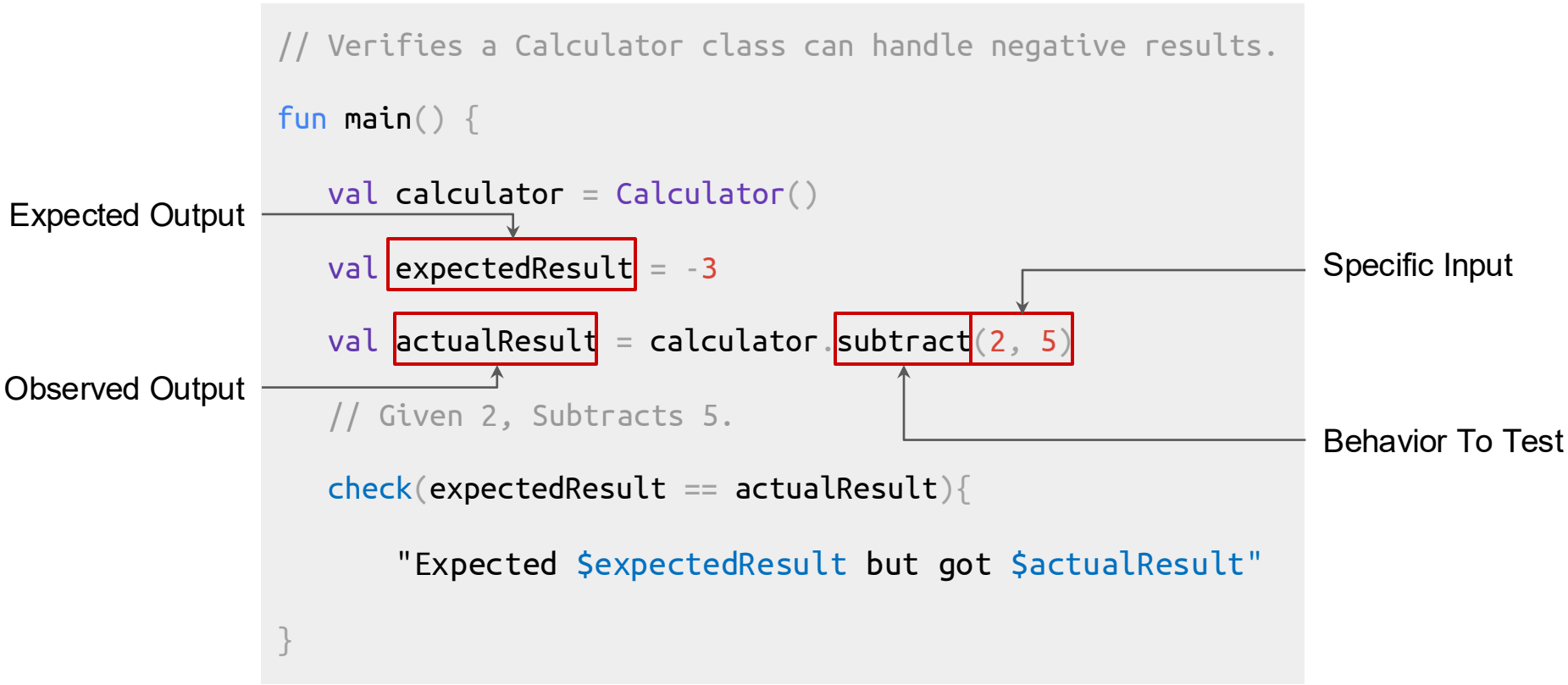
What is Testing?

- Process of evaluating software to *gain confidence* that it works as intended

- Simple test



Simple Test Example



Why Test?

- Improves the design of your system
  - Testable code is good
  - Good code is testable
- Serves as executable, up-to-date documentation
- Enables fast, high-quality releases
- Caveats
  - Edsger W. Dijkstra: “Program testing can be used to show the presence of bugs, but never to show their absence!”

Case Study: GWS before Testing

- GWS (Google Web Server): C++ application that handles requests to Google's home page
- Increase in size and complexity of projects BUT decrease in productivity
- Releases were buggier: increase in potential errors & slow search queries affecting revenue and customer trust
- Fear of developing
  - "Fear became the mind-killer. Fear stopped new team members from changing things because they didn't understand the system."
  - "Fear also stopped experienced people from changing things because they understood it all too well."

Source: https://itrevolution.com/articles/case-study-automated-testing-google/

Case Study: GWS after Testing

- Tech Lead (Bharat Mediratta) of GWS decided to institute a policy of engineer-driven, automated testing
  - All new code changes were required to include tests
  - Tests would run automatically and continuously
- Within a year
  - Drop of the # of emergency pushes by half while reaching record # of new changes every quarter
  - Today, GWS has tens of thousands of tests, and releases almost every day with relatively few customer-visible failures

Source: https://itrevolution.com/articles/case-study-automated-testing-google/

Types of Testing: Purpose

- Functional testing
  - Does my app do what it's supposed to?
- Performance testing
  - Does it do it quickly and efficiently? Does it handle high workloads?
- Usability testing
  - Can users easily accomplish their objectives with the software?
- Acceptance testing
  - Does it satisfy all the contracted requirements from the user?
- Compatibility testing
  - Does it work well on every device and API level?

Types of Testing: Scope

- Unit testing
  - Checks functionality of one method/class
- Integrated testing
  - Checks that multiple components interface correctly
- End-to-end testing
  - Checks that scenario specifications are met

Types of Testing: Environments

- Local tests
  - Tested in a development machine or server
- Instrumented tests
  - Run on actual devices (e.g.: physical Android devices)

Types of Testing: White vs. Black Box

- White box testing
  - Exploits structure within the program
  - Tests *how* the program creates output
  - Tests the logical structure of the program
- Black box testing
  - Checks whether the application functions as expected by users
  - Creates test cases based on requirements and specifications
  - Tests *what* the program does
  - We don't care *how* the application does it as long as it *outputs* the correct answer

Types of Testing: Manual vs. Automated

- Manual test
  - A human performs the tests step by step
  - Easy to perform and allows a degree of flexibility
  - Slow, subjective, and hard to scale
- Automated test
  - Tests are executed automatically via test automation frameworks, along with other tools and software.
  - Fast, objective, and scalable
  - Increased confidence in changes

Test Strategies

- Fault Injection
  - Inject exceptions, simulate failures to check if the app works in the presence of bad inputs, bad returns from libraries
- Fuzz testing
  - Multiple random inputs are thrown at your code
  - Tests an app the way it was not meant to be used
- Mutation testing
  - When an error is introduced in code, test if some tests break
  - Test the effectiveness and efficiency of test cases

Testing for Our Project

- Unit tests (week 7)
- Integrated tests (week 8)
  - Screen UI tests
  - User flow tests or navigation tests
- Usability tests (later weeks)
  - Heuristic Evaluation (HE)
  - User Acceptance Test (UAT)

Unit Tests

- Narrow-scoped tests to reduce bugs
- Tests a single function, class, or method
- Run very often and must be fast
- Does not well reflect users' perspective of the app
- Not affected by external systems

Where to Apply?

- **ViewModels / Presenters**
  - Verify UI logic, state management, and data transformations
- **Data Layer**
  - Test business logic and data flow. Replace databases or remote sources with **test doubles** (e.g., mocks, fakes, or stubs).
- **Utility Classes**
  - Validate custom helper modules such as string manipulation, math, and date utilities implemented by your team.
- **Domain Layer**
  - Test platform-independent logic including **use cases** and **interactors** that define application behavior.

How to Create Unit Tests?

- Unit tests must focus on **success**, **error** and **edge cases**
- Edge cases are uncommon scenarios that human testers and larger tests are unlikely to catch
  - Math operations using boundary conditions, inducing overflow
  - Possible network connection errors
  - Corrupted data, such as malformed JSON
  - Full storage when saving a file

What to Avoid?

- Tests to verify the correct operation of the Android framework or libraries, not your code
- Framework entry points such as activities
  - These should not have business logic
  - Leave them for instrumented tests such as UI tests

Testing Example (1/3)

- Base: Calculator App
- Github repository
  - This app does not have a UI!
- Main feature: simple addition, subtraction

```
class Calculator {

    fun add(a: Int, b: Int) = a + b
    fun subtract(a: Int, b: Int) = a - b
}
```

Testing Example (2/3)

- The *main function* (real implementation) lives in `src/main/java`
- The *testing functions* live in `src/test/java`

```kotlin
// Calculator.kt
class Calculator {

    fun add(a: Int, b: Int) = a + b
    fun subtract(a: Int, b: Int) = a - b

}
```

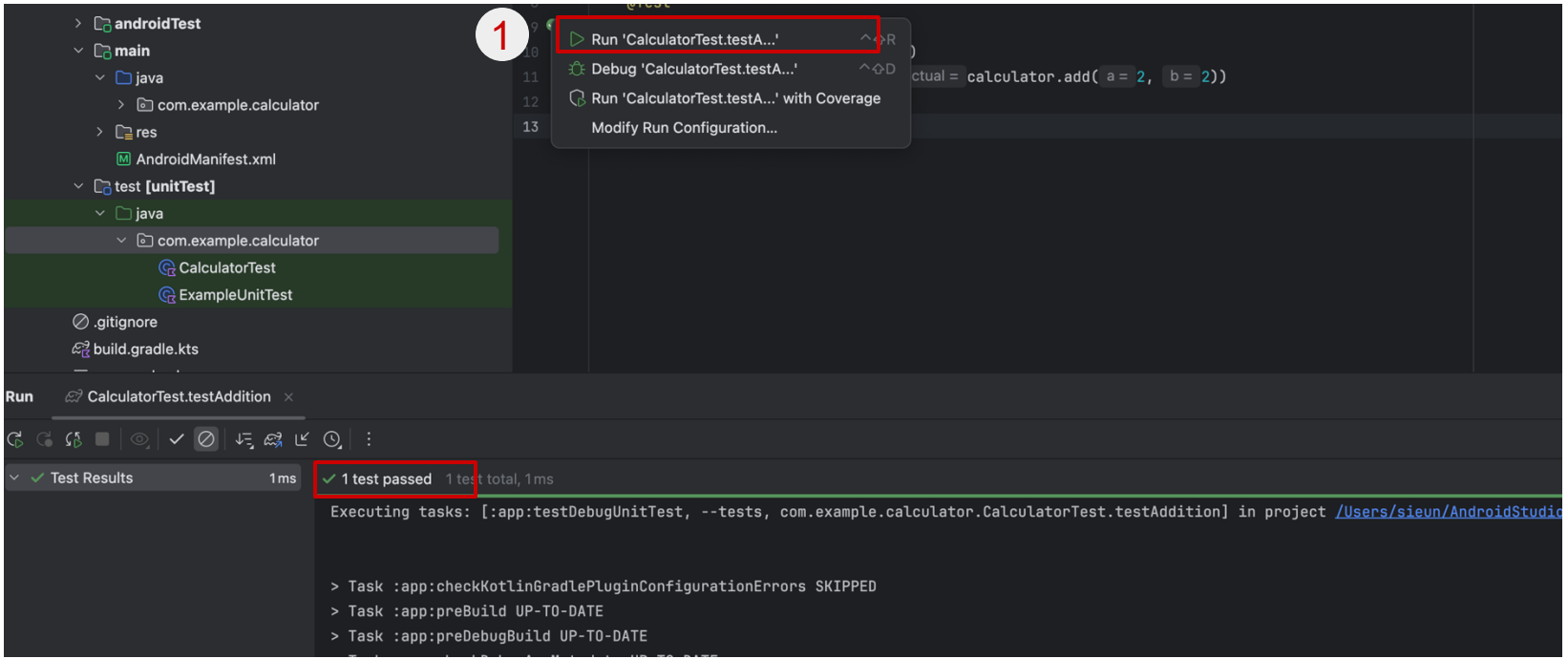
```kotlin
// CalculatorTest.kt
import org.junit.Test
import org.junit.Assert.*

class CalculatorTest {

    @Test
    fun testAddition() {
        val calculator = Calculator()
        assertEquals(4, calculator.add(2, 2))
    }
}
```

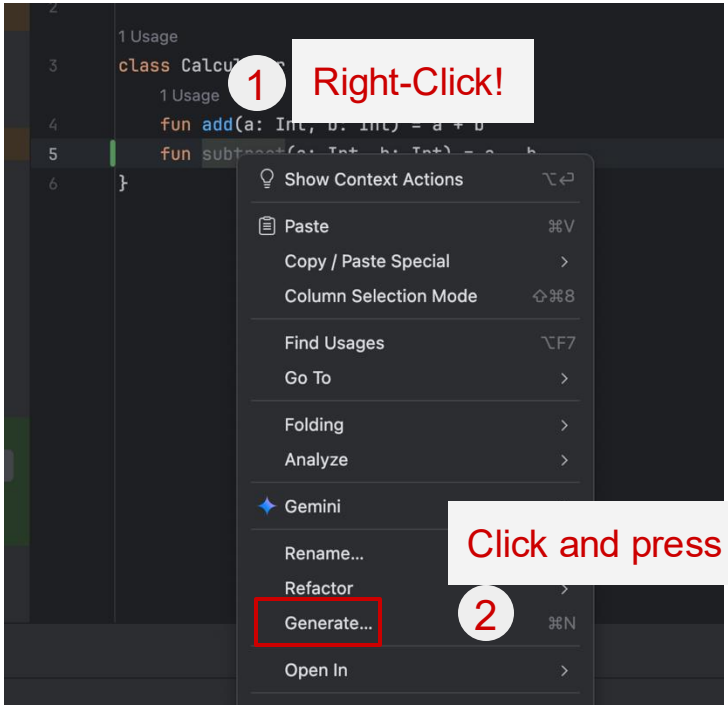
Testing Example (3/3)

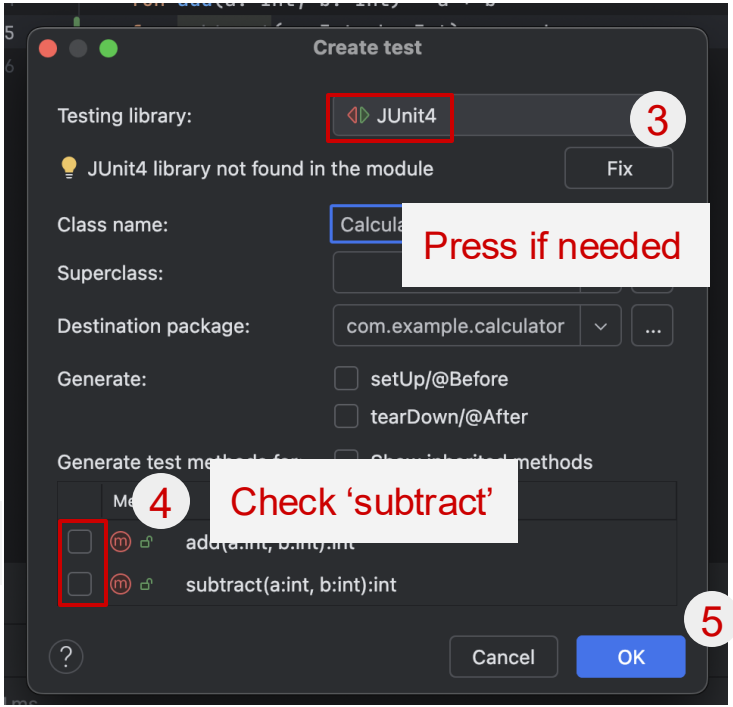
- Run your test by clicking on the green triangle. It should pass



Your First Unit Test!

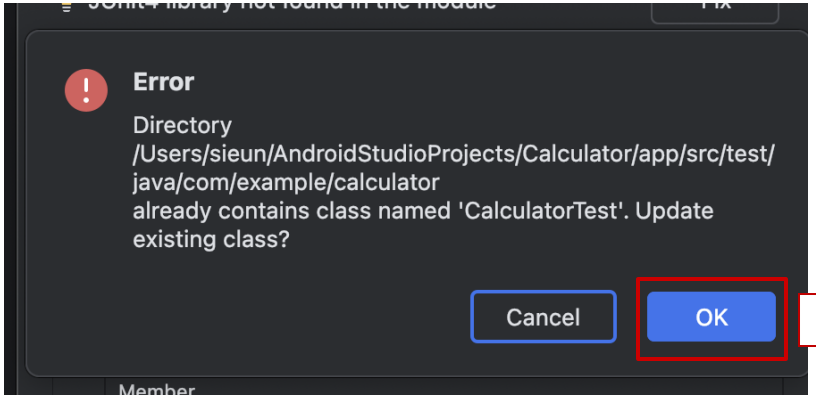
- We'll write a unit test for `Calculator/subtract`

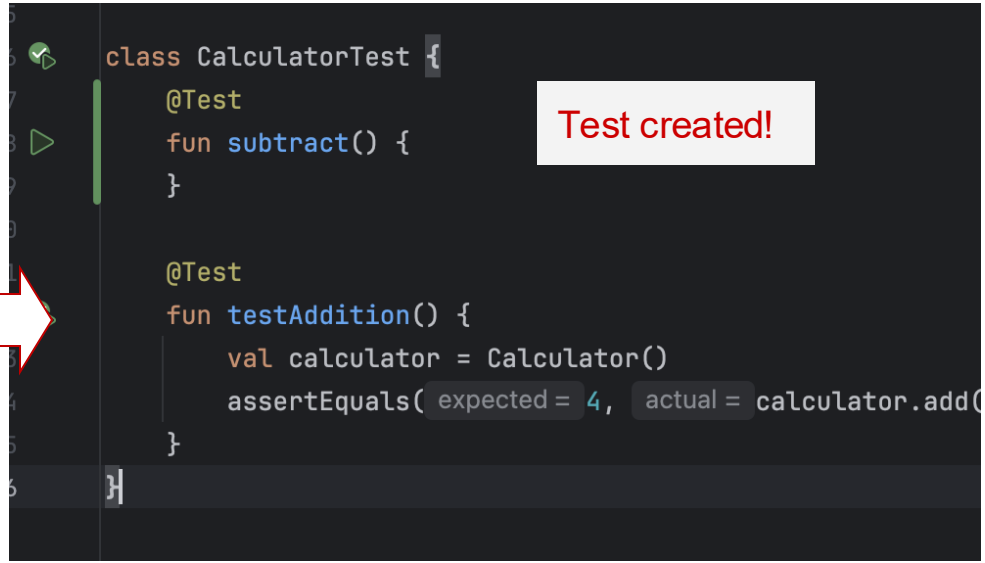




Your First Unit Test!

- We'll write a unit test for `Calculator/subtract`





Your First Unit Test!

- If the input to `subtract` is 5 and 2, it should return 3. Let's test this!
- Add a function called `subtract_five_two_return_three`
  - Naming convention used:
    - `subjectUnderTest_actionOrInput_resultState`
- Fill in using this testing mnemonic (import libraries as necessary):
  - Given / When / Then or AAA (Arrange, Act, Assert)
    - Given (Arrange): Setup the objects and app state for the test
    - When (Act): Do the actual action on the object
    - Then (Assert): Check what happens upon action

Your First Unit Test!

- Answer

```
class CalculatorTest {
    @Test
    fun subtract_five_two_return_three(){
        val calculator = Calculator()
        assertEquals(3, calculator.subtract(5,2))
    }
```

Practice
: Time for more unit tests!

Test Double

- An object or function that can replace a real implementation in a test to remove dependencies
  - Like a stunt double for an actor in an action movie

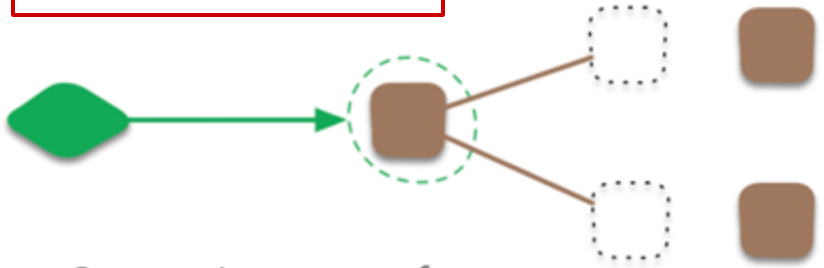
- Test doubles create controlled environment for unit tests to
  - Increase speed
  - Avoid undesired side effects during unit testing
  - Remove non-deterministic behavior

Test Double

Sociable Tests

Often the tested unit relies on other units to fulfill its behavior

Solitary Tests



Some unit testers prefer to isolate the tested unit

Source: https://martinfowler.com/bliki/UnitTest.html

Test Double Example (1/3)

- Situation: Implementing an e-commerce to process credit card services. We want to test that our method behaves correctly when the credit card is expired.

```
class PaymentProcessor (
    private val creditCardService: CreditCardService ) {
    fun makePayment (creditCard: CreditCard, amount: Money): Boolean {
        if (creditCard.isExpired()) { return false }
    val success = creditCardService.chargeCreditCard(creditCard, amount)
    return success
}}
```

- Can't use an *actual* card to test `makePayment` → use test double!

Source: https://abseil.io/resources/swe-book/html/ch13.html#example_onethree-onedot_a_credit_card_s

Test Double Example (2/3)

- Create test double class

```
class TestDoubleCreditCardService: CreditCardService {
     override fun chargeCreditCard(
         creditCard: CreditCard,
         amount: Money): Boolean {
         return true
     }
}
```

Test Double Example (3/3)

```
class PaymentProcessor(
  private val creditCardService: CreditCardService){
   ...
}
public class TestDoubleTest {
    private val paymentProcessor =
          PaymentProcessor(TestDoubleCreditCardService())

    @Test
    fun cardIsExpired_returnFalse() {
        val success = paymentProcessor.makePayment(EXPIRED_CARD, AMOUNT)
        assertThat(success).isFalse()
    }
}
```

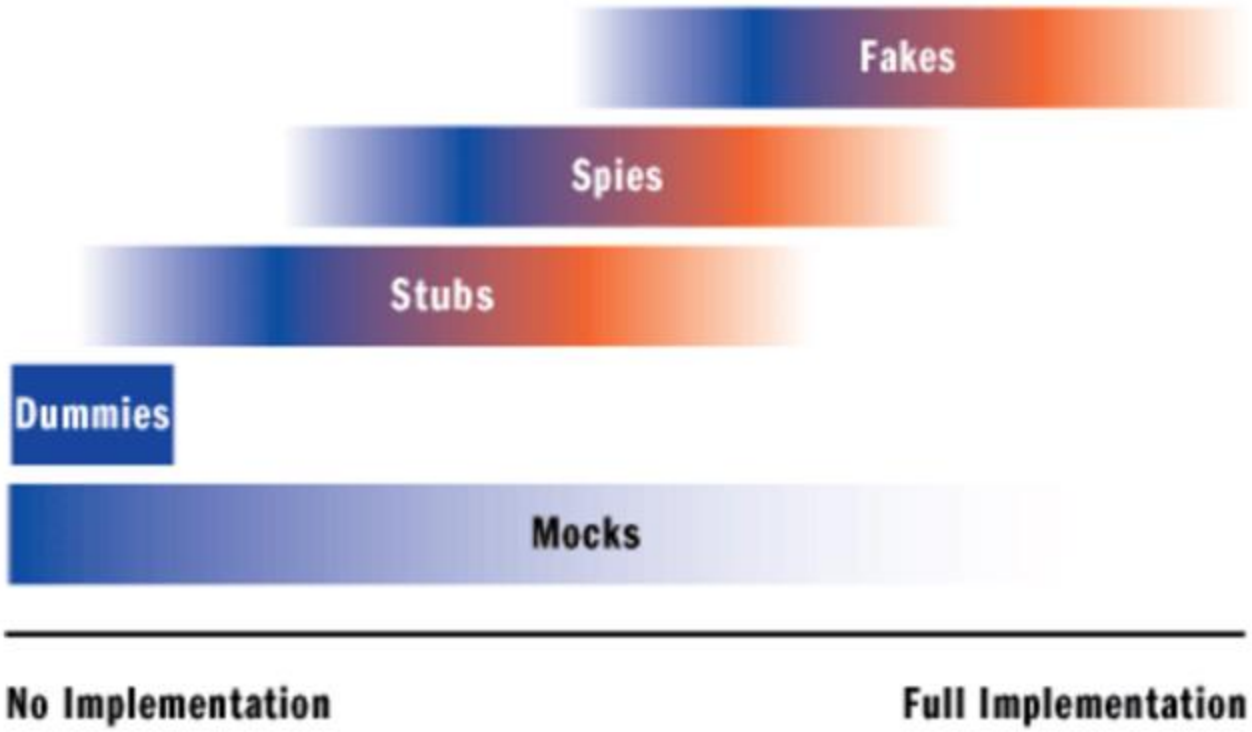
Use the test double class to remove dependency

Internally uses the test double instead of actual CreditCardService

Test Double Types

- Dummy
- Fake
- Stub
- Mock
- Spy

Test Double Spectrum



Source: https://hudi.blog/test-double/

Dummy

- Most primitive type of test double
- Empty implementation that does not perform any action
- Used when we need an instance to pass

Dummy: Example

```
interface Logger {
    fun log()
}
```

```
class LoggerDummy : Logger {
    override fun log() {
        // does nothing – dummy implementation
    }
}
```

Dummy: Example

```
class CustomerReader (
    private val logger: Logger){

    private var database: MyDataBase?= null

    // ...
}
```

```
@RunWith(MockitoJUnitRunner::class)
class ExampleUnitTest {
    // Dummy logger
    private val dummyLogger = LoggerDummy()

    // Class to be tested
    private val customerReader =
CustomerReader(dummyLogger)

    @Test
    fun happyPathScenario(){
        // ...
    }
}
```

Fake

- A much simplified but working implementation of the original function
- For instance, in-memory data structure instead of file system to store information
- Use when the productivity significantly improves

Fake: Example

```
// A fake file system.
class FakeFileSystem: FileSystem {
  // Files are stored in an in-memory map, not on disk, to avoid disk I/O in tests.
  private val files = mutableMapOf<String, String>()
  override fun writeFile(filename: String, contents: String) {
    files[filename] = contents
  }
  override fun readFile(filename: String): String {
    val contents = files[filename]
    // The real impl. will throw this exception if the file isn't found, so the fake must throw it
too.
    return contents ?: throw FileNotFoundException(fileName)
  }
}
```

Source: https://abseil.io/resources/swe-book/html/ch13.html#example_onethree-oneonedot_a_fake_file

Stub

- Fake class with no logic that comes with *pre-programmed return values*
- Use when you need a function to return a specific value to reach a certain system state

Stub: Example (1/3)

```
class CustomerReader {
    private var MyDataBase database? = null
    fun findFullName(customerID: Long): String {
        // … code here …
        val customer = database.find(customerID)
        return "${customer.firstName} ${customer.lastName}"
    }
}
```

External dependency

Stub: Example (2/3)

- Naïve approach: Pre-fill a *database* with customers
- Creates a hard dependency on a database (not isolated as unit tests should be)

```
class CustomerReaderTest {
    // Class to be tested
    private val customerReader = CustomerReader()

    // Dependency
    private val database = MyDataBase()

    @Test
    fun happyPathScenario(){
        // Prefill the database
        val sampleCustomer = Customer()
        sampleCustomer.firstName = "Susan"
        sampleCustomer.lastName = "Ivanova"
        customerReader.setDataBase(database)

        // Add to database
        database.add(sampleCustomer)

        // Test the function
        var fullName = customerReader.findFullName(1)
        assertEquals("Susan Ivanova",fullName)
    }
}
```

Stub: Example (3/3)

- Completely remove the database dependency & stub the database connection instead
- Interactions unaffected: database.find() returns a predetermined value

```
class CustomerReaderTest {
    // Class to be tested
    private val customerReader = CustomerReader()
    // Dependency --> mocked
    private val database = mock(MyDataBase::class.java)

    @Test
    fun happyPathScenario(){
        // Prefill the database
        val sampleCustomer = Customer()
        sampleCustomer.firstName = "Susan"
        sampleCustomer.lastName = "Ivanova"
        customerReader.setDataBase(database)

        // Stub the find() function to return the
sampleCustomer
        `when`(database.find(1).thenReturn(sampleCustomer)

        // Test the function
        val fullName = customerReader.findFullName(1)
        assertEquals("Susan Ivanova",fullName)
    }
}
```

Spy

- Stubs that record some information based on how they were called
- Example: an email service that records how many messages were sent

Spy: Example

```
class LoggerSpy: Logger {
    private var numberOfCalls = 0
    override fun log() {
        numberOfCalls++
    }

    public int getNumberOfCalls() {
        return numberOfCalls
    }
}
```

Mock

- Creates a fake object to check if the code behaves as expected
- Mock tests interactions/behaviours whereas stub tests states
- Example usage: to check the number or order of function calls

Mock: Example (1/3)

We want to test *notifyIfLate*() that does not return anything. How do we test it?

```
class LateInvoiceNotifier (
    private val emailSender: EmailSender,
    private val invoiceStorage: InvoiceStorage
){

    fun notifyIfLate(customer: Customer) {
        if(invoiceStorage.hasOutstandingInvoice(customer)) {
            emailSender.sendEmail(customer)
        }
    }
}
```

Mock: Example (2/3)

Check `emailSender.sendEmail` is correctly invoked depending on outstanding invoice

```
class LateInvoiceNotifierTest (
    //Class to be tested
    private lateinit var lateInvoiceNotifier: LateInvoiceNotifier
    //Dependencies (will be mocked)
    private val emailSender = mock(InvoiceStorage::class,java)
    private val invoiceStorage = mock(InvoiceStorage.class)
    //Test data
    private lateinit var sampleCustomer: Customer

    @Before
    fun setup(){
        lateInvoiceNotifier = new LateInvoiceNotifier(emailSender,invoiceStorage)
        sampleCustomer = new Customer()
        sampleCustomer.firstName = "Susan"
        sampleCustomer.lastName = "Ivanova"
    }
```

Mock: Example (3/3)

```
fun notifyIfLate(customer: Customer) {
    if(invoiceStorage.hasOutstandingInvoice(customer)) {
        emailSender.sendEmail(customer)
    }
}
```

```
    @Test
    fun lateInvoice() {
        `when`(invoiceStorage.hasOutstandingInvoice(sampleCustomer)).thenReturn(true)
        lateInvoiceNotifier.notifyIfLate(sampleCustomer)
        verify(emailSender).sendEmail(sampleCustomer)
    }

    @Test
    fun noLateInvoicePresent() {
        `when`(invoiceStorage.hasOutstandingInvoice(sampleCustomer)).thenReturn(false)
        lateInvoiceNotifier.notifyIfLate(sampleCustomer)
        verify(emailSender, times(0)).sendEmail(sampleCustomer)
    }
}
```

Verifies expected behavior happened

More on Test Doubles in Android

https://proandroiddev.com/the-definitive-guide-to-test-doubles-on-android-part-1-theory-5aa2bffb568c

Recommended Reading

https://developer.android.com/training/testing

https://developer.android.com/studio/test

Sources

- Cornell Univ 18, 19장
- Software Engineering at Google book (16,17,18)
- ESaaS chapter 8 youtube playlist
- ESaaS slide chapter 8

Thank You.
Any Questions?