**Advanced Android in Kotlin 05.1:Testing Basics**

Создать автоматически прототип теста для класса:

Встать на класс > Нажать ПКМ > Выбрать generate > Кликнуть Test

Дать имя в Class name: (дать имя)

Ок > окошко > выбрать если unit 🡪 test; если на эмуляторе, то androidTest >

Ok 🡪поместит заготовку теста в выбранную ветку, будет заготовка. (если расставить флажки, то будут заготовки для функций

testImplementation "org.hamcrest:hamcrest-all:$hamcrestVersion"

можно [Truth](https://truth.dev/), можно assertJ , но включают это сейчас в стандартную библиотеку.

Сама идея тестирования:

Дано: Единица тестирования (Ex – класс, функция, объект тестирования, viewmodel, ..)

1. Изучаем объект тестирования насколько это возможно из:

* Api - там написано, что давать и что получишь, тогда нас не интересует код.
* Из комментариев к объекту без кода или с кодом.
* Из описания функции (заголовок) можно угадать
* Лезем в код и читаем сам Java/Kotlin пытаемся понять, что дать и что получить.
* От программера матерно с перчислением всего, что он не сделал сверху.
* Если все выше указанное не помогло, то ДО слакс и т.п. спросить.  
  Пример, который рассматриваем очень примитивный, реальная функция может тянуть всякую хрень из 10-ти мест, кроме параметров которые ей передаются и изменять данные в хреновой туче мест. ( это .опа)
* Поэтому новый стиль программирования требует что бы функция пользовалась ТОЛЬКО параметрами в круглых скобках и больше ничем. И требует что бы результат ( то что она изменяет) возвращался только после :
* Но это почти никем и никогда не выполняется.

2. Надо объекту подготовить тестовые данные:

* В лучшем случае параметры, которые в скобках
* В более плохом случае, все данные вокруг, куда он еще залезает
* В самом худшем случае, это вся программа, тогда это сквозной тест.
* Для этого создаем экземпляры объектов, программера или своих
* Заполняем эти объекты данными
* Создаем заглушки вместо модулей, которые он вызывает. Моски, спаи и т.п.
* Если нам нужен Android, то по возможности используем робоэлектрик (он иммулирует андроид без эмелятора и телефона)
* Если не получается робоэлектрик, то загоняем тест на эмулятор или телефон и андроид будет создавать тестовую среду.
* При использовании Hilt – DI внедрение зависимостей. Все становится на порядок сложнее и на порядок проще, т.к. мы заполняем зависимости, а заберет их модуль сам. См. большой модуль 69/34

3. Мы наконец-то зовем тестируемый модуль и ему все это вручаем,   
что подготовили в п.2 и начинаем отлаживать тест на каком-то одном наборе данных, что занимает от 30 мин до 3-х дней.

Когда получен результат от первого теста, то дается второй, третий и весь тестовый набор.

4. Тест прогоняется на всем тестовом наборе и становится пассед. И если он за собой тянет полное покрытие, то считается, что объект работает правильно.   
Надо создать тестовый набор с полным покрытием, что бы было все протестировано.   
А это может быть 10-ки тестов и может повлечь подготовку данных.

5. Дописывается комментарий к тесту, он вставляется в автоматизированные старты, закрываются акты и получаются деньги.

6. Программист меняет объект, на чем тест становится краш, на этом все разваливается и начинай с п.1 по 5

7. Заключается договор на поддержание тестов в рабочем состоянии. Изменение их и т.п.

\*\* И что-нибудь складываешь в портфолио.

Все вышеперечисленное слабо относится к TDD там пункты выполняется снизу вверх и ты должен быть постановщиком, если постановщик не выдается тебе полное api

Особенность в TDD, что отлаживать тесты не на чем и сначала все тесты будут красными. Пока не будет создан модуль.

Тогда задача программера, что бы тесты были зелеными.

Итого:

Тест это: AAA – «Упорядочить, Действовать, Утвердить»

Note that the" Arrange, Act, Assert" (AAA) testing mnemonic is a similar concept.

TDD:

GWT- Given, When, Then - Учитывая, когда, тогда / Учитывая, когда, то / Взял, Когда, Тогда

Подготовить данные, Вызвать объект, Сравнить полученные результаты

Чтобы исправить код и написать тесты, вы будете использовать разработку через тестирование. Разработка через тестирование выполняется следующим образом.

1. Напишите тест, используя структуру Given, When, Then и имя, соответствующее соглашению.
2. Подтвердите, что тест не прошел.
3. Напишите минимальный код, чтобы пройти тест.
4. Повторите для всех тестов!

Вместо того, чтобы начинать с исправления ошибки, вы начнете с написания тестов. Затем вы можете подтвердить, что у вас есть тесты, защищающие вас от случайного повторного появления этих ошибок в будущем.

1. Если есть пустой список ( emptyList()), тогда оба процента должны быть 0f.
2. Если при загрузке задач произошла ошибка, список будет null, и оба процента должны быть 0f.
3. **Запустите** тесты и убедитесь, что они **не работают** :
4. */\*\*  
    \* Получив ошибку, что пустой список не дает 0/0 при TDD, я тестер пишу программеру задание  
    \* исправь объект, что бы проходил тест.  
    \* Я программер, прочитав это задание - иду в этот объект  
    \* и начинаю ее исправлять, что бы тест стал зеленым.  
    \*/  
   /\*\*  
    \* Программер изменил код сам запустил не изменяя тест, получил зеленое, сдал код и побежал за денежкой.  
    \* Вывод не тестер пишет за программером, а программер пишет за тестером  
    \* Можно программера иметь junior+, а не сеньора-.   
    \* Но, как всегда обратная сторона медали, что тестер должен быть сеньором, или над ним стоит сеньор, а тестер его прекрасно понимает.  
    \* Тестер описывает всю структуру тестами не имея объектами.  
    \* Побочное явление - имена придумывает не junior, а тестер или сеньёр  
    \*/*

Следуя TDD и сначала написав тесты, вы помогли убедиться, что:

* Новая функциональность всегда связана с тестами; таким образом, ваши тесты служат документацией того, что делает ваш код.
* Ваши тесты проверяют правильность результатов и защищают от ошибок, которые вы уже видели.

Чтобы узнать больше о разработке через тестирование в Android, вы можете посмотреть [раздел Разработка через тестирование на Android с помощью библиотеки поддержки тестирования Android (Google I / O '17)](https://youtu.be/pK7W5npkhho?t=222) (обратите внимание, что библиотека поддержки тестирования Android была заменена на [AndroidX Test,](https://developer.android.com/training/testing/set-up-project" \t "_blank) которую вы узнаю позже).

**AndroidX.Test - это пакет framework тестовых библиотек   
на сегодня 50шт. (1.3.0 – 1.4.0 – alpha 05)**

**Прежде всего он отдает эммулированные куски Androida такие как контекст, фрагмент и т.д. Это же делает робоэлектрик как-то по другому для других частей.**

Включая анотации, Core (ядро), Junit, Espresso (много), Thrush, Rule, Runner, Filters, jang, и д.р. в т.ч. uiAutomator

От нас требуется подключать те из 50-ти которые нам нужны для тестов. Не все сразу.

AndroidX.Test на сегодня не заменяет собой Mockito, Roboelectric и другие сторонние библиотеки, даже Junit4 и не заменяет многие базовые библиотеки, но много уже туда втянуто.

В оставшейся части кодовой лаборатории вы узнаете, как писать тесты для двух классов Android, которые являются общими для большинства приложений - [ViewModel](https://developer.android.com/topic/libraries/architecture/viewmodel" \t "_blank)и [LiveData](https://developer.android.com/topic/libraries/architecture/livedata" \t "_blank).

Вы начинаете с написания тестов для платформы TasksViewModel.

Вы собираетесь сосредоточиться на тестах, которые имеют всю свою логику в модели представления и не полагаются на код репозитория. Код репозитория включает асинхронный код, базы данных и сетевые вызовы, что увеличивает сложность тестирования. На данный момент вы собираетесь избежать этого и сосредоточиться на написании тестов для функциональности ViewModel, которые напрямую не тестируют что-либо в репозитории.

**ViewModel** -

**Model-View-ViewModel** (MVVM) — шаблон проектирования архитектуры приложения. (вместо MVC/MVP)  
В рамках этого шаблона есть понятие ViewModel.

Model – fragment

View – xml

VM – **ViewModel**

В наше Jetpack Android полном случае:

К **ViewModel** обращается сверху фрагмент (и XML напрямую)

А ViewModel обращается к репозиторию

В чистейшем коде с идеальной архитектурой – это все. Но есть дополнительные модули, которые хватает фрагменты или саму **ViewModel** и сама может куда-то обращаться, куда не предусмотрели

\* При hilt там все еще строже будет

По Китти следует действовать так: Мы слушаем кнопку + Листенером onClickListener и дальше когда кнопка нажата, то слушатель перехватывает и идет действие

Теперь рекомендуется, но не обязательно, делать так:

Нажатие на кнопку – порождает событие Event c параметром на что нажали. А все слушают, когда появится очередное событие, если событие его, то они его обрабатывают.

После этого они еще обнуляют событие или нет. Это зависит от постановки.

Очень не видно, очень запутано, но это везде.

Не брать пока это в голову. ДО еще не разобрался. У нас в ToDo этого еще нет.

Что такое AndroidX Test?

AndroidX Test - это набор библиотек для тестирования. Он включает классы и методы, которые предоставляют вам версии компонентов, таких как Приложения и Действия, которые предназначены для тестов. В качестве примера, этот код, который вы написали, является примером функции AndroidX Test для получения контекста приложения.

ApplicationProvider.getApplicationContext()

Одним из преимуществ API-интерфейсов AndroidX Test является то, что они созданы для работы как для локальных, так и для инструментальных тестов. Это приятно, потому что:

Вы можете запустить тот же тест, что и локальный тест, или инструментальный тест.

Вам не нужно изучать разные API тестирования для локальных и инструментальных тестов.

Например, поскольку вы написали свой код с использованием библиотек AndroidX Test, вы можете переместить свой TasksViewModelTestкласс из testпапки в androidTestпапку, и тесты по-прежнему будут выполняться.

Он getApplicationContext()работает немного по-разному в зависимости от того, запускается ли он как локальный или инструментальный тест:

Если это инструментальный тест, он получит фактический контекст приложения при загрузке эмулятора или подключении к реальному устройству.

Если это локальный тест, он использует смоделированную среду Android.

**Что такое Робоэлектрик?**

Смоделированная среда Android, которую AndroidX Test использует для локальных тестов, предоставляется Robolectric . Robolectric - это библиотека, которая создает смоделированную среду Android для тестов и работает быстрее, чем загрузка эмулятора или запуск на устройстве.

@RunWith(AndroidJUnit4::class) **делать?**

Средство выполнения [**тестов**](https://github.com/junit-team/junit4/wiki/Test-runners) - это компонент JUnit, запускающий тесты. Без средства выполнения тестов ваши тесты не запускались бы. JUnit предоставляет средство запуска тестов по умолчанию, которое вы получаете автоматически. [@RunWith](https://github.com/junit-team/junit4/wiki/Test-runners#runwith-annotation)заменяет этот тестовый исполнитель по умолчанию.

Средство выполнения [AndroidJUnit4](https://developer.android.com/training/testing/junit-runner#kotlin)тестов позволяет AndroidX Test запускать ваш тест по-разному в зависимости от того, являются ли они инструментальными или локальными тестами.

От ДО: задача добавление новой задачи

Нам поставлена цель проверить работает ли функция добавить новую задачу класса ViewModel

Что бы вызвать функцию, мы должны создать экземпляр этого класса ViewModel

Но что бы создать экземпляр класса ViewModel мы должны передать параметр application, т.е. окружение Android (он должен быть создан на эмуляторе)

На Эмуляторе любой тест стартует тяжело и долго, поэтому мы обращаемся к фреймфорку AndroidX.Test и просим его создай нам фейковый контекст application, что он и делает попутно зачем-то использую робоэлектрик

Когда он нам контекст создал и отдал ApplicationProvider.getApplicationContext(), то мы создаем экземпляр ViewModel для тестирования передавая ему этот контекст (фейковый)

Ура! Мы теперь имеем экземпляр для тестов да еще не в эмуляторе, а в JVM Windows

Теперь мы можем вызывать эту функцию для тестов.

Для проверки LiveDataрекомендуется сделать две вещи:

Использовать InstantTaskExecutorRule

Обеспечить LiveDataнаблюдение

Шаг 1. Используйте InstantTaskExecutorRule

InstantTaskExecutorRuleэто правило JUnit . Когда вы используете его с @get:Ruleаннотацией, он вызывает InstantTaskExecutorRuleзапуск некоторого кода в классе до и после тестов (чтобы увидеть точный код, вы можете использовать сочетание клавиш Command + B для просмотра файла).

Это правило запускает все фоновые задания, связанные с компонентами архитектуры, в одном потоке, чтобы результаты теста выполнялись синхронно и в повторяющемся порядке. Когда вы пишете тесты, включающие тестирование LiveData, используйте это правило!

От ДО: Android многопоточный, т.е. выполняется сразу несколько задач.  
Когда какая задача выполнится – зависит от разных, в т.ч. внешних условий.  
Например Internet медленнее отвечает. И задача опроса выполнится позже чем задача обращения Sql базы к телефону. Или наоборот.  
Для тестов это почти недопустимо, т.к. тесты все время будут проходить по разному.  
Поэтому в тестовых случаях все выполняют в одном потоке, одно за другим и в одинаковом порядке, что бы иметь устойчивый результат.

А если надо тестировать другой порядок, то это тест №2.

LiveData (живые данные)- наблюдаемый класс держателя данных.  
Переменные бывают разных типов например: Int, …. (базовые)

Переменные бывают типа какого-нибудь класса и тогда они указательный экземпляр этого класса.

Классов всяких разных море и туча и еще столько же. В пакетах Android других пакетах  
Мы сами их создаем, а потом создаем переменные типа этого класса – указатель на экземпляр.

Android пришло в голову создать такой класс и обозвать его LiveData, что они и сделали.  
Android ребята объявили его очень важным, нужным и центральным в Jetpack и на пьедестал его. Это теперь очень важный класс. Мы его должны изучить и им пользоваться. Если мы программисты Jetpack

Понятия теперь о переменных которые на него смотрят.

Если в обычную переменную (инт мы кладем какое-то значение, то старое стирается и новое там хранится и все) Когда мы обратимся к новой переменной, то она отдаст нам новое значение.

В переменную LiveData мы кладем значение, оно там лежит, НО еще она сообщает всем, кто за ней наблюдает о том, что в нее положили (она получила) новое значение.

Из этого соображения понятно, что мало того, что надо объявить эту переменную LiveData (тогда она будет обычной переменной, но просто неудобной), но еще надо объявить ее наблюдателей, минимум 1 или больше. (Если нет ни одного наблюдателя, то нет смысла ее объявлять).

**Это начало декларативного программирования:**

Я объявляю декларацию и больше ничего не делаю.

Пример: В Экселе пишу формулу (объявляю декларацию) в ячейку С заносить сумму ячейки А и ячейки B. Больше я ничего не делаю. Юзер в ячейке А взял и написал 8 – ячейка С изменилась. ТАК КАК когда в ячейку А вбили новое число, то она разоралась ячейке С, которая за ней наблюдает (т.к. в формуле ячейки С участвует ячейка А) , что у нее изменилось число. Ячейка С это услышала и выполнила формулу, которая в ней стоит с этим новым числом, а у меня в ячейке С появилась новая сумма.

Примечания:

У LiveData - Дикий синтаксис: вместо a=5, Пишем a.value=5 и т.д.

Итого: что бы просто проверить функцию новая задача в этой ViewModel мы должны:

1. Создать класс ViewModel с поддельным контекстом AndroidX+Робоэлектрик

2. Вызвать стандартное добавление задачи

3. Поставить слушателя – наблюдателя с помощью изврата LiveDataTestUtil.kt

4. Получить от него ответ слышит он или нет изменения списка задач

5. Проверить его ответ на нул. Т.е. не слышит.

6. Решить прошел тест или нет.

7. Не забыть поставить все в один поток методом rule

## 12. Резюме

Эта кодовая лаборатория охватывала:

* Как запускать тесты из Android Studio.
* Разница между локальными ( test) и инструментальными тестами ( androidTest).
* Как писать локальные модульные тесты с использованием [JUnit](https://junit.org/junit4/" \t "_blank) и [Hamcrest](http://hamcrest.org/" \t "_blank) .
* Настройка тестов ViewModel с помощью [библиотеки](https://developer.android.com/training/testing/set-up-project) тестов [AndroidX](https://developer.android.com/training/testing/set-up-project" \t "_blank) .

13. Узнать больше

Образцы:

Официальный образец тестирования - это официальный образец тестирования, основанный на том же приложении TO-DO Notes, используемом здесь. Концепции в этом примере выходят за рамки того, что описано в трех тестовых кодовых таблицах.

Демо Sunflower - это основной образец Android Jetpack, который также использует библиотеки тестирования Android.

Курс Udacity:

Разработка приложений для Android с помощью Kotlin

Документация разработчика Android:

Руководство по архитектуре приложения

JUnit4

Hamcrest

Библиотека робототехнических испытаний

Библиотека тестов AndroidX

Базовая тестовая библиотека компонентов архитектуры AndroidX

исходные наборы

Тест из командной строки

Видео:

Разработка через тестирование на Android с помощью библиотеки поддержки тестирования Android (Google I / O'17) .

Создание тестируемых приложений для Android (Google I / O'19)

Другой:

Модульное тестирование LiveData и другие распространенные проблемы наблюдаемости

https://developer.android.com/codelabs/advanced-android-kotlin-training-testing-basics#12