**fakes and mocks: Продвинутый Android в Kotlin 05.2:Введение в тестовые двойники и внедрение зависимостей**

1. Продолжаем в проекте из 5.1

2. Исправил все библиотеки – поднял до последних

3. Исправил FAB: 

private fun setupFab() {  
 viewDataBinding.addTaskFab.setOnClickListener **{** navigateToAddNewTask() **}** /\* сбоит  
 activity?.findViewById<FloatingActionButton>(R.id.add\_task\_fab)?.let {  
 it.setOnClickListener {  
 navigateToAddNewTask()  
 }  
 }\*/  
}

Эта вторая тестовая кодовая лаборатория посвящена тестовым двойникам: когда их использовать в Android и как их реализовать с помощью внедрения зависимостей, шаблона Service Locator (это ручное устаревшее, теперь Hilt, в копии 69/34 такая же, но с Hilt) и библиотек. Поступая так, вы научитесь писать:

Модульные тесты репозитория

Интеграционные тесты фрагментов и моделей просмотра

Тесты навигации по фрагментам

Что ты узнаешь

Как спланировать стратегию тестирования

Как создавать и использовать тестовые двойники, а именно фейки и моки

Как использовать ручное внедрение зависимостей на Android для модульных и интеграционных тестов

Как применить шаблон Service Locator

Как тестировать репозитории, фрагменты, модели просмотра и компонент навигации

Вы будете использовать следующие библиотеки и концепции кода:

* runBlocking и runBlockingTest
* FragmentScenario
* Эспрессо
* Mockito

Что ты будешь делать

* Напишите модульные тесты для репозитория, используя двойной тест и внедрение зависимостей.
* Напишите модульные тесты для модели представления, используя двойной тест и внедрение зависимостей.
* Напишите интеграционные тесты для фрагментов и их моделей представления, используя среду тестирования пользовательского интерфейса Espresso.
* Напишите тесты навигации, используя Mockito и Espresso

3. Концепция: стратегия тестирования.

В этой кодовой лаборатории вы узнаете, как тестировать репозитории, просматривать модели и фрагменты с помощью тестовых двойников и внедрения зависимостей. Прежде чем вы углубитесь в то, что это такое, важно понять рассуждения, которые будут определять, что и как вы будете писать эти тесты.

В этом разделе рассматриваются некоторые общие рекомендации по тестированию применительно к Android.

Пирамида тестирования

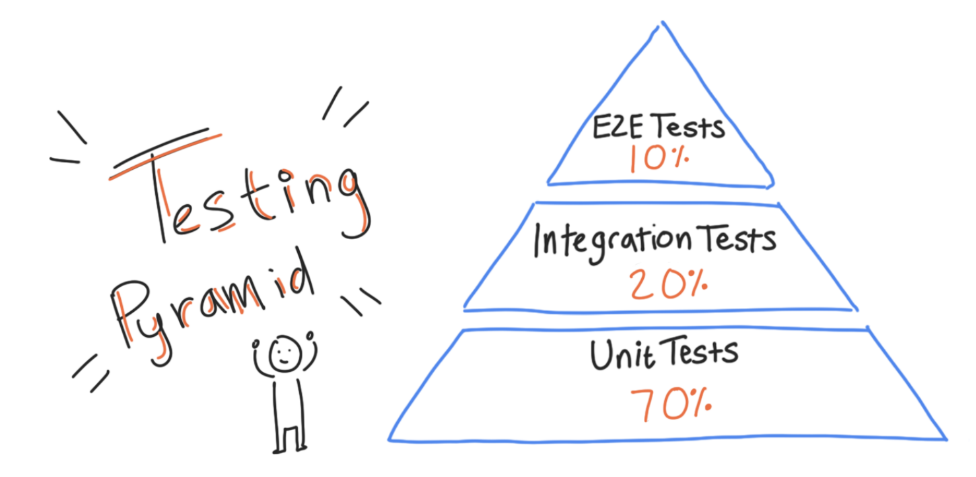
Когда вы думаете о стратегии тестирования, есть три связанных аспекта тестирования:

* **Область действия -** Какую часть кода затрагивает тест? Тесты могут выполняться одним методом, во всем приложении или где-то посередине.
* **Скорость.** Насколько быстро проходит тест? Скорость тестирования может варьироваться от миллисекунд до нескольких минут.
* **Верность -** Насколько "реальный" тест? Например, если часть тестируемого кода должна выполнить сетевой запрос, действительно ли тестовый код выполняет этот сетевой запрос или подделывает результат? Если тест действительно разговаривает с сетью, это означает, что у него более высокая точность. Компромисс заключается в том, что выполнение теста может занять больше времени, может привести к ошибкам, если сеть не работает, или может быть дорогостоящим в использовании.

Между этими аспектами есть неизбежные компромиссы. Например, скорость и точность - это компромисс: чем быстрее тест, тем меньше точность, и наоборот. Один из распространенных способов разделить автоматизированные тесты - на эти три категории:

* Модульные **тесты** - это узконаправленные тесты, которые выполняются в одном классе, обычно в одном методе этого класса. Если модульный тест не проходит, вы можете точно знать, где именно в коде возникла проблема. У них низкая точность, поскольку в реальном мире ваше приложение включает в себя гораздо больше, чем выполнение одного метода или класса. Они достаточно быстрые, чтобы запускаться каждый раз, когда вы меняете свой код. Чаще всего это тесты, запускаемые локально (в testисходном наборе). **Пример:** тестирование отдельных методов в моделях представлений и репозиториях.
* **Интеграционные тесты** - они проверяют взаимодействие нескольких классов, чтобы убедиться, что при совместном использовании они ведут себя должным образом. Один из способов структурирования интеграционных тестов - заставить их протестировать одну функцию, например возможность сохранить задачу. Они тестируют больший объем кода, чем модульные тесты, но по-прежнему оптимизированы для быстрой работы по сравнению с полной точностью. Их можно запускать локально или как инструментальные тесты, в зависимости от ситуации. **Пример:** тестирование всех функциональных возможностей отдельной пары фрагмент и модель представления.
* **Сквозные тесты (E2e)** - **проверьте** сочетание работающих вместе функций. Они тестируют большие части приложения, тщательно моделируют реальное использование и поэтому обычно работают медленно. Они обладают высочайшей точностью и говорят вам, что ваше приложение действительно работает как единое целое. По большому счету, эти тесты будут инструментальными тестами (в androidTestисходном наборе). **Пример:** запуск всего приложения и одновременное тестирование нескольких функций.

Предлагаемая пропорция этих тестов часто представлена ​​пирамидой, при этом подавляющее большинство тестов являются модульными.

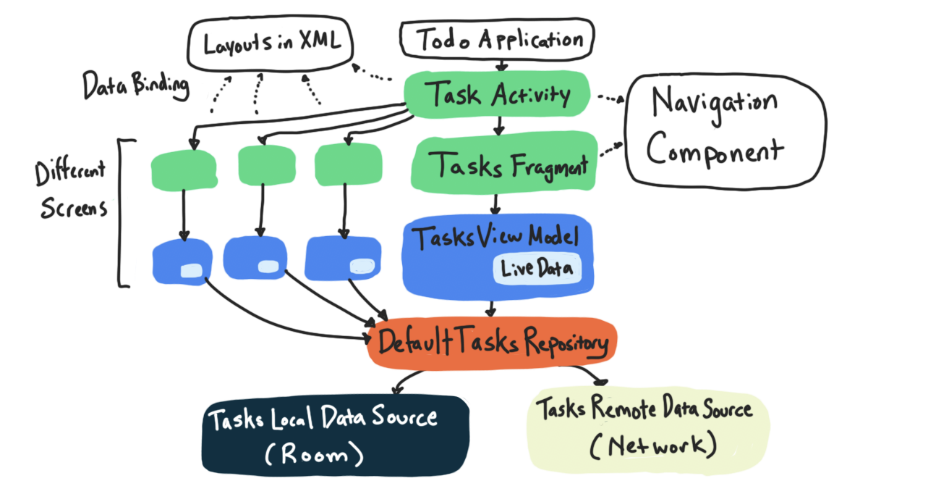


Ознакомьтесь с [Основами тестирования](https://developer.android.com/training/testing/fundamentals) в документации для разработчиков Android, чтобы более глубоко погрузиться в эти концепции и то, как они соотносятся с Android.

Архитектура и тестирование

Ваша способность тестировать приложение на всех уровнях пирамиды тестирования по своей сути связана с **архитектурой** вашего **приложения** . Например, приложение с *очень* плохой архитектурой может поместить всю свою логику в один метод. Возможно, вы сможете написать для этого сквозной тест, поскольку эти тесты, как правило, тестируют большие части приложения, но как насчет написания модульных или интеграционных тестов? Когда весь код находится в одном месте, сложно протестировать только код, относящийся к отдельному модулю или функции.

Лучшим подходом было бы разбить логику приложения на несколько методов и классов, позволяя тестировать каждую часть изолированно. Архитектура - это способ разделения и организации кода, который упрощает модульное и интеграционное тестирование. Приложение TO-DO, которое вы будете тестировать, следует определенной архитектуре:



В этом уроке вы увидите, как тестировать части вышеупомянутой архитектуры в надлежащей изоляции:

1. Во- первых вы **модульное тестирование** в **хранилище** .
2. Затем вы будете использовать тестовый двойник в модели представления, который необходим для **модульного тестирования** и **интеграционного тестирования** модели представления.
3. Далее вы научитесь писать **интеграционные тесты** для **фрагментов и их моделей представления** .
4. Наконец, вы научитесь писать **интеграционные тесты** , включающие **компонент навигации** .

Сквозное тестирование будет рассмотрено в следующем уроке.

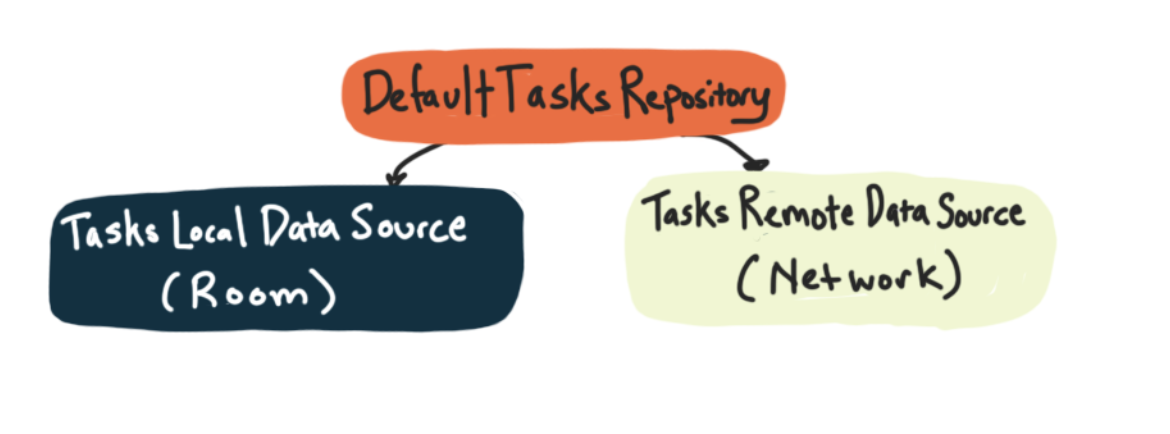
4. Задача: создать поддельный источник данных.

Когда вы пишете модульный тест для части класса (метода или небольшого набора методов), ваша цель - **протестировать только код этого класса** .

Тестирование только кода в определенном классе или классах может быть сложной задачей. Давайте посмотрим на пример. Откройте data.source.DefaultTaskRepositoryкласс в mainисходном наборе. Это репозиторий для приложения и класс, для которого вы будете писать модульные тесты в следующий раз.

Ваша цель - протестировать только код этого класса. Тем не менее, работа DefaultTaskRepositoryзависит от других классов, таких как LocalTaskDataSourceи RemoteTaskDataSource. Другой способ сказать это , что LocalTaskDataSourceи RemoteTaskDataSourceявляются **зависимости** от DefaultTaskRepository.

Таким образом, каждый метод DefaultTaskRepositoryвызывает методы классов источников данных, которые, в свою очередь, вызывают методы других классов для сохранения информации в базе данных или связи с сетью.



Например, взгляните на этот метод в DefaultTasksRepo.

Это одна из функций этого класса, которая вызывается когда мы кликаем по задаче. Она обращается к базе данных и просит у нее дай эту задачу, и эта функция отдает эту задачу наверх тому, кто ее вызвал. Exp из ViewModel

// suspend – это значит, что функция содержит корутины (стартует параллельные процессы) или вызывает функции, которые содержат корутины

Suspend – приостановка без захвата и без блокировки – очень неудачное название хитросложного механизма: Основное, что он стартует функцию в другом потоке. Или много функций в разных потоках (Thread), но и это не верно, т.к. он стартует не в потоках, а в нитях одного потока. На самом деле там хитрый механизм старт-стопа с возвратом. Suspend ДОЛЖЕН вызываться только из корутин или другого Suspend, но тогда тот Suspend должен быть вызван из корутины.   
Примечание: корутины в отличие от старта потока делают следующие вещи:  
1. Снимают снимок с состояния программы  
2. Стартует параллельные вычисления в нити, а не в потоке.  
3. По окончанию вычисления восстанавливают состояния из снимка 1.  
Благодаря использования корутин, а не потоков исчезают колбеки, сокращается кол-во кода, лучше передаются данные между параллельными процессами. Используется моща Котлин.  
Примечание: Лучше не задумываться о том, что транслятор Kotlin корутины все равно превратит в потоки Java.  
Пример: Есть задача как можно быстрее израсходовать бензин в разных машинах, но все машины тащатся по одной полосе, т.к. сужение. Тут сужение заканчивается и каждой машине выделили свою полосу, они все рванули вперед и каждая съела свой бензин. У одной машины закончился бензин. Позвонит организатору, что у нее там-то кончился бензин. (это и есть колбэк).

suspend fun getTasks(forceUpdate: Boolean = false): Result<List<Task>> {

// forceUpdate: Boolean = false – обновлять ли задачи из сети в Room перед вопросом – выбирает вызывающий. Надо обновлять - true или нет - false.

        if (forceUpdate) {  
            try {  
                updateTasksFromRemoteDataSource()  
            } catch (ex: Exception) {  
                return Result.Error(ex)  
            }  
        }

// tasksLocalDataSource.getTasks() – обращается к Room дай список всех задач, а получив список передает вызывающему.  
        return tasksLocalDataSource.getTasks()  
    }

Корутины нужны на запись, а на чтение не очень.

В данном случае: getTasks – она хоть и suspend, но корутины не содержит и следовательно ее протестировать будет проще, чем правда suspend функцию. О чем коделабс мне и рассказывает.

На ней suspend стоит с дуру.

Test Doubles – тестовые дубли

|  |  |
| --- | --- |
| **Fake Фальшивый** | A test double that has a "working" implementation of the class, but it's implemented in a way that makes it good for tests but unsuitable for production.  Двойник теста, имеющий «рабочую» реализацию класса, но реализованный таким образом, что он удобен для тестов, но не подходит для производства. |
| **Mock Насмехаться** | A test double that tracks which of its methods were called. It then passes or fails a test depending on whether it's methods were called correctly.  Двойник теста отслеживает, какие из его методов были вызваны. Затем он проходит или не проходит тест в зависимости от того, правильно ли были вызваны его методы. |
| **Stub Заглушка** | A test double that includes no logic and only returns what you program it to return. A StubTaskRepository could be programmed to return certain combinations of tasks from getTasks for example.  Тестовый двойник, который не включает логики и возвращает только то, что вы его запрограммировали. Например, A StubTaskRepositoryможно запрограммировать, чтобы возвращать определенные комбинации задач getTasks. |
| **Dummy** Дурачок | A test double that is passed around but not used, such as if you just need to provide it as a parameter. If you had a NoOpTaskRepository, it would just implement the TaskRepository with **no** code in any of the methods.  Тестовый двойник, который передается, но не используется, например, если вам просто нужно указать его в качестве параметра. Если вы имели NoOpTaskRepository, было бы просто реализовать TaskRepositoryс **не** кода в любом из методов. |
| **Spy Шпион** | A test double which also keeps tracks of some additional information; for example, if you made a SpyTaskRepository, it might keep track of the number of times the addTask method was called.  Тестовый дублер, который также отслеживает некоторую дополнительную информацию; например, если вы создали SpyTaskRepository, он может отслеживать количество вызовов addTask метода. |

Для получения дополнительной информации о тестовых двойниках ознакомьтесь с [разделом «Тестирование в туалете: знай свои тестовые пары»](https://testing.googleblog.com/2013/07/testing-on-toilet-know-your-test-doubles.html) .

Наиболее распространенными тестовыми двойниками, используемыми в Android, являются **Fakes** и **Mocks** .

В этой задаче вы собираетесь создать FakeDataSourceтестовый двойник для модульного теста, DefaultTasksRepositoryотделенный от реальных источников данных.

**Резюме**

1. Модульные тесты - это узконаправленные тесты, которые обычно тестируют один класс.
2. **DefaultTaskRepository**сложен для модульного тестирования, потому что он имеет две сложные **зависимости** : **TasksLocalDataSource**и **TasksRemoteDataSource**.
3. Чтобы решить эту проблему, вы сделаете **тестовые дубли** для замены **TasksLocalDataSource**и **TasksRemoteDataSource**при тестировании.
4. Это позволит вам писать модульные тесты, которые содержат только тестовый **DefaultTaskRepository**код.

Мы наконец создали фейковый класс на замену руму и инету, но   
как теперь заставить наш тестируемый объект **DefaultTaskRepository** обратиться к нашему фейку, а не к реальному коду рума и нетвока.  
В общем случае это невозможно без исправления кода программиста, если вызовы вшиты в тестируемый объект.

Поэтому по новому стилю программирования вшивать вызовы в объекты и функции нельзя.  
Надо передавать их через список параметров тестируемому объекту, а внутри объекта вызывать их через переданный параметр. В этом случае счастье тестировщика. Это и есть DI (внедрение зависимостей)

Если программа написана с учетом DI, то тестировщик из своего тестового модуля может создать класс или вызвать функцию тестируемого объекта. Передав им в качестве предусмотренных параметров свой фейк сделанный тестировщиком. И тогда тестируемый объект при обращении к зависимостям будет обращаться к фейку тестировщика.

Тестируем Побережского в офисе:

Ему должен позвонить адм, а он должен дать распоряжение на склад. Мы не знаем правильно ли Побережский действует и не хотим портить рабочий процесс. Поэтому мы применяем внедрение зависимостей, т.е. записную книжку Поб мы подменяем фейковой. А Хоменко у нас будет фейк за склад и адм. Тогда действия след:

ЛВ (тестовый модуль) дает распоряжение Поб - реши вопрос.

Хоменко по фейковому номеру адм звонит к Поб и сообщает, что перегорели лампочки, но Поб глядя в свой телефон видит, что звонит адм.

Поб берет свой телефон и звонит по строчке склад (но мы его подменили) и он попадает опять к Хоменко. И он дает складу указание «…..» После действий Поб дает отчет ЛВ (в тестовый модуль):  
а) все компы отпущены, деньги собраны, потерь нет – ЛВ Тест пройден

Б) шеф усе пропало, все разакрали – ЛВ тест не пройден,

Если б) то программер переделывает Поб.

Подмена телефонной книге Поб – есть подсовывание фейка через внедрение зависимостей.

Если у Поб книжка в телефоне, то это можно сделать.   
А если он звонит по Памяти, и телефоны написаны на руке, то подсунуть фейк нельзя и модуль надо переделывать.

На сейчас соорудили стол Хоменко с одним телефоном и с чем надо, что бы Поб дурить.  
Теперь будем подсовывать записную книжку.

Мы создали фейковый репозиторий, который надо подсунуть вместо локального и сетевого.

Дальше создаем тестовый класс, который будет вызывать, подсовывать, тестировать и проверять.

В этом тестовом классе создаем фейковый репозиторий (Before), который надо подсунуть. А для них создаем поддельный список задач. Фейковые списки создаем прямо наверху.

Наконец в этом тестовом классе пишем наш тест, где вызываем getTasks(true), т.е. мы обращаемся к тестируемому репозиторию с просьбой дай список задач и перед этим обнови их в сети, но тестовый объект обратиться к рум и интернету через фейк, который мы ему подсунули. А фейк подсунет ему список задач, который мы в тестовом классе подготовили. Остается спросить тестовый объект возвращает нам тот список, который мы ему подсунули(ок) или что-то другое(fail)?

На этом он работает.

Так же можно протестировать getTasks на обращение к фейку локального репозитория.

ВНИМАНИЕ!!

Будет отдельная морока с каталогом

testImplementation "org.jetbrains.kotlinx:kotlinx-coroutines-test:$coroutinesTestVersion"

т.к. версия 1.42 они выбросили runBlockTest и т.д. и работает 1.35

Вторая проблема @ExperimentalCoroutinesApi а остальное потом в 6-ом и дальше уроках

Протестируем локальный фейк

Создадим еще один тест с другим именем

Вызовем опять getTasks(false), а по api этого метода, если ему передать фалсе, то он должен взять с локальной базы данных.

Полученный ответ сравню с фейковой локальным списком задач, который подсовываю фейковому локальному репозиторию. Если совпадают, то Ок, если нет, fail

Так делать нехорошо, это только для отладки теста, его надо закомментить.

Закомментируем.

6. Задача: настроить поддельный репозиторий.

Будем тестировать ViewModel, а для этого теперь создадим поддельный репозиторий.

Поддельный репозиторий создается через создание с него интерфейса.

Первым шагом к использованию внедрения зависимостей конструктора является создание общего интерфейса, разделяемого между подделкой и реальным классом.

Создаем интерфейс с объекта который надо подделать, т.е. с DefaultTasksRepository в рабочей части программы, это мы работаем за программиста, а не за тестера. Преобразуем программный код, что бы внедрить там зависимости. Только после этого тестер может подсунуть создать фейк.

Отступление из документации: DI (внедрение зависимостей) бывает:  
 а) ручное, т.е. зависимость выносим в параметры и изменяем вызов. См. 5.2 начало  
 б) Сервисе локатор – жуткая технология по созданию этого сервис локатера,   
 прописыванию зависимостей и втыканию в программу – автоматизированно см.5.2 конец  
 в) Даггер пакет фреймворк очень мощный, все может, бошку свернешь  
 г) Hilt = лайт даггер = упрощение сервис локатора, зато принят гуглом в Jetpack,  
 поддерживается Android Studio обладает своей системой тестирования см. Hilt ToDo app  
 69/34

Д) могут быть еще какие-то другие пакеты зависимостей, которые нам не интересны

При создании этого Refactor->Extract - она делает слепок с класса и кладет его в интерфейс.

А еще она исходный опирает на этот только созданный интерфейс, а все его функции к fun добавляет override. По идее этот класс + интерфейс должен работать так же как старый класс.

Теперь имея интерфейс в программной части в тестовой части создаем fake опираясь на этот интерфейс. – вот это дело тестера.

Создали фейковый репозиторий на основании выделенного интерфейса.

Что бы его встроить во ViewModel, ViewModel должна принимать ссылку на репозиторий через параметры, а у нее репозиторий создается внутри – изменяем.

Во ViewModel убираем создание репозитория и ставим его в получаемые параметры ViewModel.

Но тогда изменяется создание ViewModel во фрагменте и нам пришлось создать фабрику для ViewModel и указать во фрагменте создавать ViewModel через эту фабрику, что мы и сделали.

Все эти телодвижения не должны нарушить правильную работу программы. – это рефакторинг.

Заумно: мы произвели рефакторинг ViewModel c целью DI в нее репозитория.

Теперь будем сооружать тестовый пример TasksViewModelTest на фейковом репозитории FakeTestRepository внедряя его через DI в тестируемую реальную ViewModel:

Используя внедрение зависимостей конструктора, вы удалили DefaultTasksRepositoryзависимость as и заменили ее на вашу FakeTestRepositoryв тестах.

Повторили внедрение зависимости для ViewModelDetail и теперь эту ViewModel тоже можно тестировать с тем же фейковым репозиторием. Только надо написать тест.

8, Затем вы напишете интеграционные тесты для проверки взаимодействия фрагмента и модели представления

<https://developer.android.com/codelabs/advanced-android-kotlin-training-testing-test-doubles#7>

8. Задача: запустить фрагмент из теста.

Затем вы напишете интеграционные тесты для проверки взаимодействия фрагмента и модели представления. Вы узнаете, правильно ли обновляет код вашей модели представления ваш пользовательский интерфейс. Для этого вы используете

* шаблон ServiceLocator
* библиотеки Espresso и Mockito

**Интеграционные тесты** проверяют взаимодействие нескольких классов, чтобы убедиться, что при совместном использовании они ведут себя должным образом. Эти тесты можно запускать либо локально ( testисходный набор), либо как инструментальные тесты ( androidTestисходный набор).

В вашем случае вы будете брать каждый фрагмент и писать интеграционные тесты для модели фрагмента и представления, чтобы проверить основные функции фрагмента.

Ваш тест должен как загрузить TaskDetailFragment(что вы сделали), так и подтвердить, что данные были загружены правильно. Почему нет данных? Это потому, что вы создали задачу, но не сохранили ее в репозиторий.

Он у вас есть FakeTestRepository, но вам нужен способ заменить реальный репозиторий фальшивым репозиторием для вашего фрагмента . Вы сделаете это дальше!

## 9. Задача: создать ServiceLocator.

Есть проблема: когда мы тестировали отдельный модуль ViewModel, то мы создавали его в тестовом классе и имели возможность подпихнуть под него фейковый (передать ему в параметры) фейковый репозиторий. Дальше было все прекрасно. Тест спрашивает ViewModel, она обращается к фейковому репозиторию который подсунули, фейковый репозиторий отдает назад подготовленный нами фейковый ответ, ViewModel его получает обратно, выполняет свою функцию, которую вызвали, возвращает ответ тесту.  
тест сравнивает ответ с ожидаемым и ставит галку или крест.

А теперь интеграционный тест, т.е. будем тестировать фрагмент плюс ViewModel.

Создаем тестовый класс, он создает фрагмент (через Сценарий).  
И здесь проблема – тестировщик не может создать ViewModel со своим фековым репозиторием, хотя во ViewModel есть внедрение зависимостей DI. Т.к. ее создает сам фрагмент, когда его стартовал тестировщик. И эта задача не разрешимая всеми методами, которые пока нам рассказали.

И тут выходит на сцену программный изврат ServiceLocator: его задача мы создаем фрагмент, фрагмент создает ViewModel, а ServiceLocator при этом создании подпихивает туда фейк репозиторий.

\*\*\* Т.к. ServiceLocator это зубы вырви, то теперь применяется Hilt, то же не сахар, но не такой кошмар.

ServiceLocator – должен быть встроен в код тестируемой программы заранее программистом (так же как внедрение зависимостей или интерфейс репозитория)

class TodoApplication : Application() {  
  
 // Запросить ServiceLocator создать один экземпляр TasksRepository и вернуть нам ссылку на него,  
 // которой мы будем пользоваться дальше   
 val taskRepository: TasksRepository  
 get() = ServiceLocator.provideTasksRepository(this)

предложить исправление taskRepository (4-ре раза используется) – не нужен, можно использовать var tasksRepository (5 раз используется) : TasksRepository из ServiceLocator

В коделабе нас сначала заставляют создать ServiceLocator и из него вызвать рабочий репозиторий = равно убедиться, что ничего не изменилось.

Затем нас просят создать фейковый репозиторий для интеграционного теста в androidTest.

Нам предлагают скопировать текст кода без разбора, т.к. это почти копия фейкового репозитория из ветки тест. Но там дописаны кое-какие реализации.

(при написании в общем-то можно снять копию из ветки тест в ветку андроид тест. Но еще лучше, наморочиться с Gradle и использовать один на двоих. Если одинаковый годится и для тестирования ViewModel (в тест) и для интеграционного фрагмент плюс ViewModel (androidTest)

Можно потом его разобрать и расписать его реализации в этом фейковом репозитории.

В рабочей программе по правилам архитектуры репозиторий и база данных создается один и потом на него только выдаются ссылки на этот же много раз и это правильно.

Это не годится для тестов, т.к. мы запишем что-то одним тестом, а потом другим еще запишем, тогда тесты будут не изолированы друг от друга.

Пример: Если учитель рассказывает урок, то он все время пишет дальше. Но если он вызывает к доске для учеников, решать каждому свой пример по очереди, то перед след учеником надо стереть, что там нарешал предыдущий, а не давать ему дописывать от предыдущего.

Из-за этого мы полезли в сервис локатор и добавили функцию очищения репозитория и базы данных (доски). Которую будем вызывать только из теста, чтобы только ставится аннотация.

На мой взгляд, это глупость, я бы так не делал. Я бы в тесте в @Before вызвал бы создание нового фейкового репозитория и новой базы данных. А в @After их утилизацию. И она бы по правилам Junit перед каждым тестом создавала бы новую базу, а выполнив тест за собой подчищала бы. И мне не пришлось бы лазить в сервис локатор программиста.