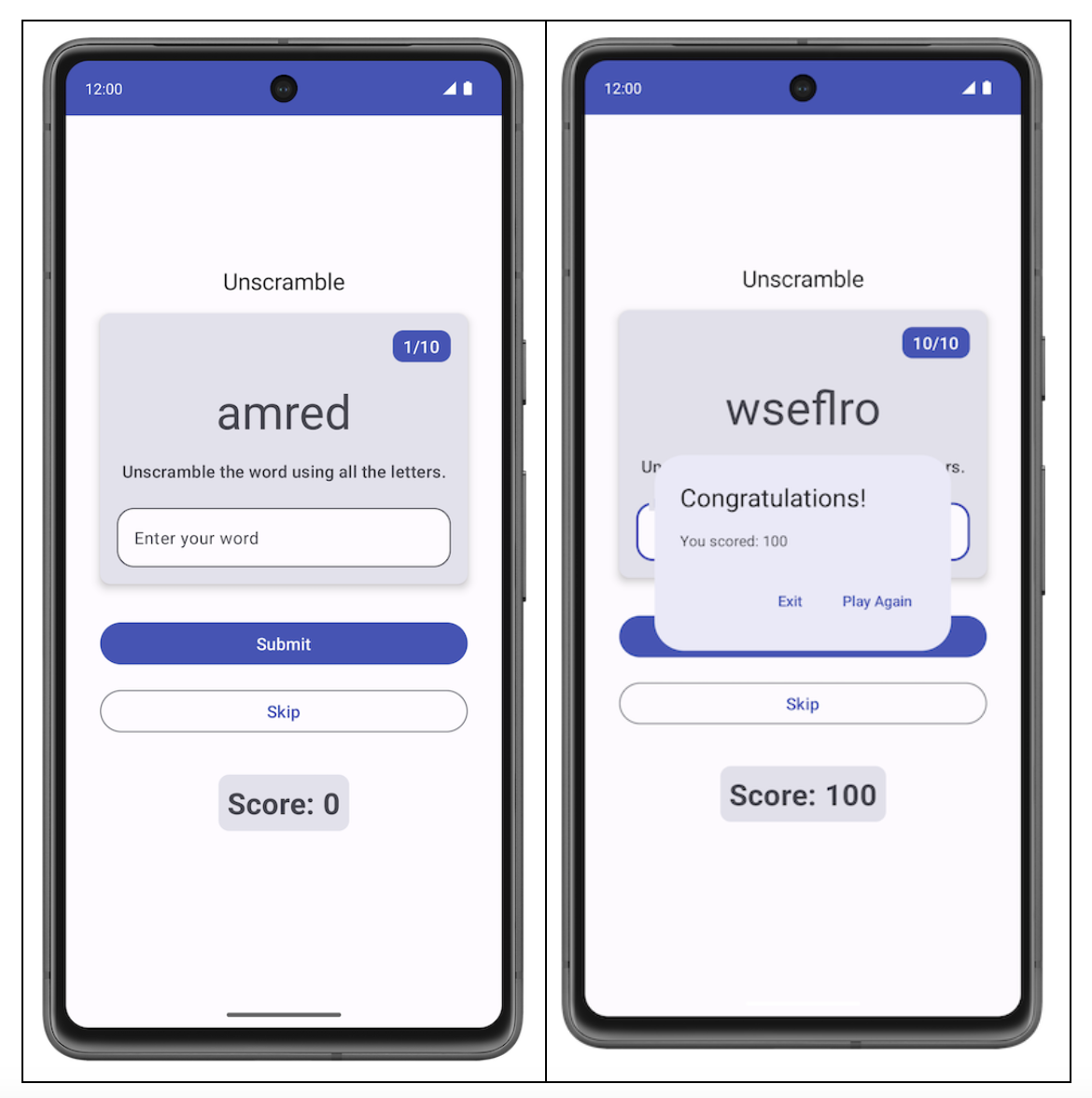
27 Написание модульных тестов для ViewModel

[1. Прежде чем начать](https://developer.android.com/codelabs/basic-android-kotlin-compose-test-viewmodel?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-compose-unit-4-pathway-1%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-compose-test-viewmodel#0)

Эта лаборатория кода научит вас писать модульные тесты для тестирования ViewModelкомпонента. Вы добавите модульные тесты для игрового приложения [Unscramble](https://github.com/google-developer-training/basic-android-kotlin-compose-training-unscramble.git) . Приложение Unscramble — это забавная словесная игра, в которой пользователи должны угадывать зашифрованное слово и зарабатывать очки за правильное угадывание. На следующем изображении показан предварительный просмотр приложения:



В лабораторной работе [«Написание автоматических тестов»](https://developer.android.com/codelabs/basic-android-kotlin-compose-write-automated-tests?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-compose-unit-2-pathway-3%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-compose-write-automated-tests#0) вы узнали, что такое автоматизированные тесты и почему они важны. Вы также узнали, как реализовать модульные тесты.

Ты выучил:

* Автоматизированное тестирование — это код, который проверяет точность другого фрагмента кода.
* Тестирование — важная часть процесса разработки приложения. Постоянно запуская тесты вашего приложения, вы можете проверить его функциональное поведение и удобство использования, прежде чем выпустить его публично.
* С помощью модульных тестов вы можете тестировать функции, классы и свойства.
* Локальные модульные тесты выполняются на вашей рабочей станции, что означает, что они выполняются в среде разработки без необходимости использования устройства Android или эмулятора. Другими словами, на вашем компьютере запускаются локальные тесты.

Прежде чем продолжить, убедитесь, что вы выполнили лабораторные работы [по написанию автоматических тестов](https://developer.android.com/codelabs/basic-android-kotlin-compose-write-automated-tests?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-compose-unit-2-pathway-3%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-compose-write-automated-tests#0) и [ViewModel и State в Compose](https://developer.android.com/codelabs/basic-android-kotlin-compose-viewmodel-and-state) .

Предварительные условия

* Знание Kotlin, включая функции, лямбды и составные элементы без сохранения состояния.
* Базовые знания о том, как создавать макеты в Jetpack Compose.
* Базовые знания Material Design.
* Базовые знания о том, как реализовать ViewModel.

Что вы узнаете

* build.gradle.ktsКак добавить зависимости для модульных тестов в файл модуля приложения
* Как создать стратегию тестирования для реализации модульных тестов
* Как писать модульные тесты с использованием JUnit4 и понимать жизненный цикл тестового экземпляра
* Как запускать, анализировать и улучшать покрытие кода

Что ты построишь

* Юнит-тесты для игрового приложения [Unscramble](https://github.com/google-developer-training/basic-android-kotlin-compose-training-unscramble.git)

Что вам понадобится

* Последняя версия Android Studio

Получить стартовый код

Для начала загрузите стартовый код:

[file\_downloadСкачать zip](https://github.com/google-developer-training/basic-android-kotlin-compose-training-unscramble/archive/refs/heads/viewmodel.zip)

Альтернативно вы можете клонировать репозиторий GitHub для кода:

$ git clone https://github.com/google-developer-training/basic-android-kotlin-compose-training-unscramble.git

$ cd basic-android-kotlin-compose-training-unscramble

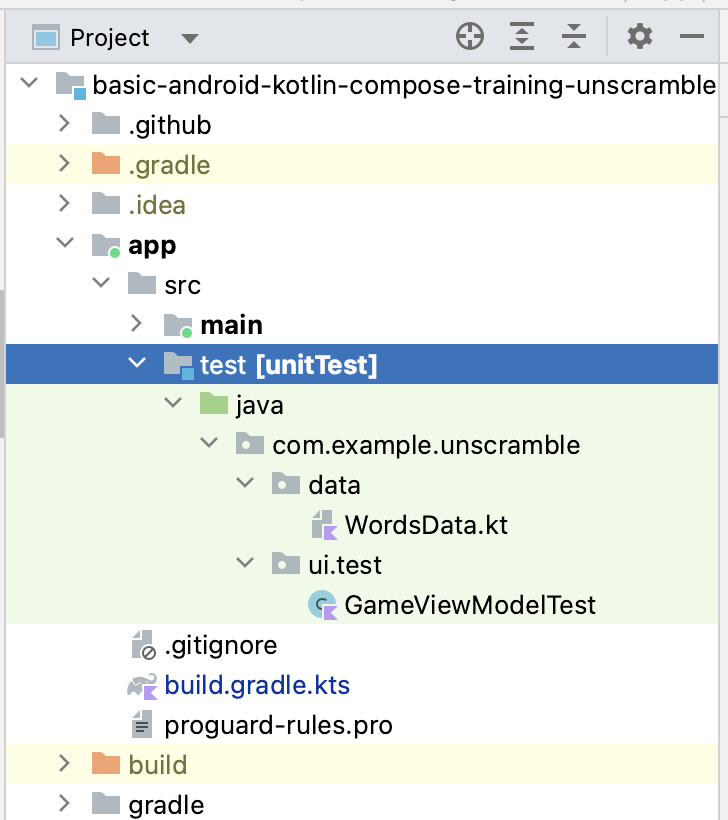
$ git checkout viewmodel

**Примечание.** Стартовый код находится в viewmodelветке скачанного репозитория.

Вы можете просмотреть код в [Unscramble](https://github.com/google-developer-training/basic-android-kotlin-compose-training-unscramble/tree/viewmodel)репозитории GitHub.

[2. Обзор стартового кода](https://developer.android.com/codelabs/basic-android-kotlin-compose-test-viewmodel?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-compose-unit-4-pathway-1%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-compose-test-viewmodel#1)

В модуле 2 вы научились размещать код модульного теста в наборе исходного кода **теста** , который находится в папке **src** , как показано на следующем рисунке:



Стартовый код имеет следующий файл:

* WordsData.kt**:** Этот файл содержит список слов, которые будут использоваться для тестирования, и getUnscrambledWord()вспомогательную функцию для получения расшифрованного слова из зашифрованного слова. Вам не нужно изменять этот файл.

**Примечание.** Свойство allWordsв файле набора исходного кода теста WordsData.ktбудет использоваться файлом GameViewModel. При запуске тестов это заменяет allWordsсвойство, доступное в файле набора исходного кода приложения WordsData.kt.

В следующем модуле вы познакомитесь с новой техникой, называемой внедрением зависимостей, которая способствует слабой связи и помогает использовать различные ресурсы при выполнении тестов.

[3. Добавьте тестовые зависимости](https://developer.android.com/codelabs/basic-android-kotlin-compose-test-viewmodel?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-compose-unit-4-pathway-1%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-compose-test-viewmodel#2)

В этой лаборатории кода вы используете платформу JUnit для написания модульных тестов. Чтобы использовать платформу, вам необходимо добавить ее как зависимость в build.gradle.ktsфайл модуля вашего приложения.

Вы используете implementationконфигурацию, чтобы указать зависимости, необходимые вашему приложению. Например, чтобы использовать ViewModelбиблиотеку в своем приложении, необходимо добавить зависимость к androidx.lifecycle:lifecycle-viewmodel-compose, как показано в следующем фрагменте кода:

dependencies {  
  
    ...  
    implementation("androidx.lifecycle:lifecycle-viewmodel-compose:2.6.1")  
}

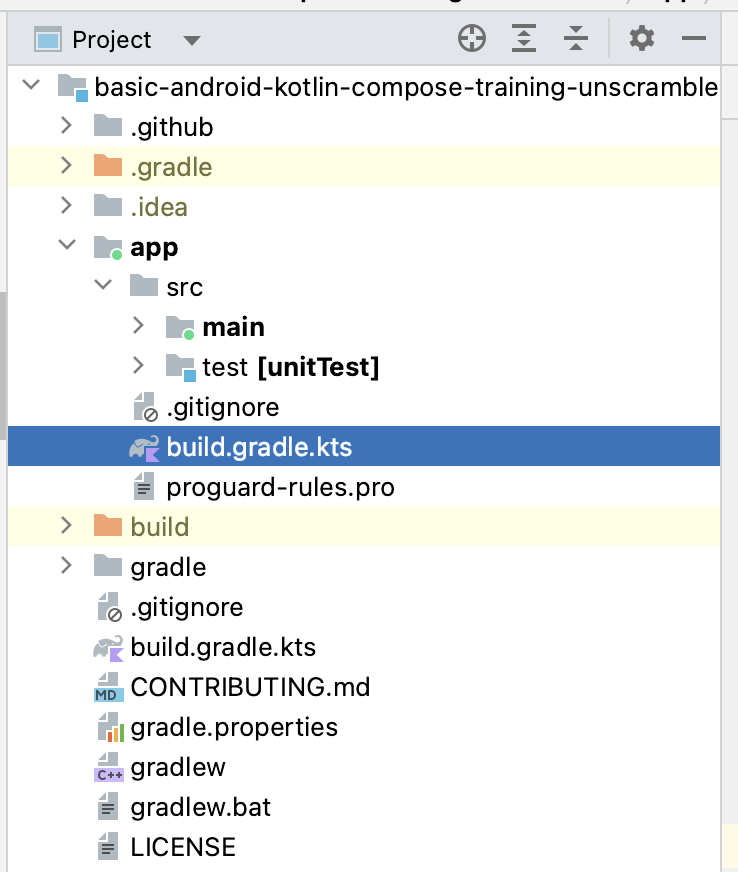
Теперь вы можете использовать эту библиотеку в исходном коде своего приложения, а студия Android поможет добавить ее в созданный файл пакета приложения (APK). Однако вы не хотите, чтобы код модульного теста был частью вашего APK-файла. Тестовый код не добавляет никаких функций, которые мог бы использовать пользователь, и код также влияет на размер APK. То же самое касается зависимостей, необходимых вашему тестовому коду; вам следует держать их отдельно. Для этого вы используете testImplementationконфигурацию, которая указывает, что конфигурация применяется к локальному исходному коду теста, а не к коду приложения.

**Примечание.** Каждое приложение Android компилируется и упаковывается в один файл, называемый файлом пакета приложения (APK), который включает в себя весь код приложения, ресурсы, активы и файл манифеста. Для удобства файл пакета приложения часто называют APK и имеет расширение **.apk** . Устройства под управлением Android используют этот файл для установки приложения.

Чтобы добавить зависимость в проект, укажите конфигурацию зависимости (например, implementationили testImplementation) в блоке зависимостей вашего build.gradle.ktsфайла. Каждая конфигурация зависимостей предоставляет Gradle различные инструкции о том, как использовать зависимость.

Чтобы добавить зависимость:

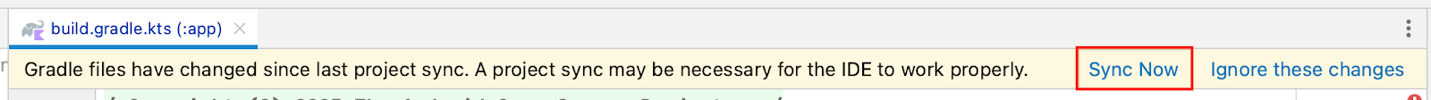
1. Откройте файл appмодуля build.gradle.kts, расположенный в appкаталоге на панели **«Проект»** .



1. Внутри файла прокрутите вниз, пока не найдете dependencies{}блок. Добавьте зависимость, используя testImplementationконфигурацию для junit.

plugins {  
    ...  
}  
  
android {  
    ...  
}  
  
dependencies {  
    ...  
    testImplementation("junit:junit:4.13.2")  
}

1. На панели уведомлений в верхней части файла **build.gradle.kts** нажмите **«Синхронизировать сейчас»** , чтобы завершить импорт и сборку, как показано на следующем снимке экрана:



**Составьте спецификацию материалов (BOM)**

Compose BOM — рекомендуемый способ управления версиями библиотеки Compose. Спецификация позволяет вам управлять всеми версиями библиотеки Compose, указав только версию спецификации.

Обратите внимание на раздел зависимостей в файле appмодуля build.gradle.kts.

// No need to copy over  
// This is part of starter code  
dependencies {  
  
   // Import the Compose BOM  
    implementation (platform("androidx.compose:compose-bom:2023.06.01"))  
    ...  
    implementation("androidx.compose.material3:material3")  
    implementation("androidx.compose.ui:ui")  
    implementation("androidx.compose.ui:ui-graphics")  
    implementation("androidx.compose.ui:ui-tooling-preview")  
    ...  
}

Соблюдайте следующее:

* Номера версий составной библиотеки не указаны.
* Спецификация импортируется с использованиемimplementation platform("androidx.compose:compose-bom:2023.06.01")

Это связано с тем, что сама спецификация содержит ссылки на последние стабильные версии различных библиотек Compose, поэтому они хорошо работают вместе. При использовании спецификации в вашем приложении вам не нужно добавлять какую-либо версию в сами зависимости библиотеки Compose. Когда вы обновляете версию спецификации, все используемые вами библиотеки автоматически обновляются до новых версий.

Чтобы использовать спецификацию с библиотеками тестирования (инструментальными тестами), вам необходимо импортировать файлы androidTestImplementation platform("androidx.compose:compose-bom:xxxx.xx.xx"). Вы можете создать переменную и повторно использовать ее для implementationи, androidTestImplementationкак показано.

// Example, not need to copy over  
dependencies {  
  
   // Import the Compose BOM  
    implementation(platform("androidx.compose:compose-bom:2023.06.01"))  
    implementation("androidx.compose.material:material")  
    implementation("androidx.compose.ui:ui")  
    implementation("androidx.compose.ui:ui-tooling-preview")  
      
    // ...  
    androidTestImplementation(platform("androidx.compose:compose-bom:2023.06.01"))  
    androidTestImplementation("androidx.compose.ui:ui-test-junit4")  
  
}

**Примечание** . Составление спецификации предназначено только для библиотек Compose, а не для других библиотек, таких как androidx.lifecycleбиблиотека жизненного цикла.

Большой! Вы успешно добавили тестовые зависимости в приложение и узнали о спецификации. Теперь вы готовы добавить несколько модульных тестов.

## [4. Стратегия тестирования](https://developer.android.com/codelabs/basic-android-kotlin-compose-test-viewmodel?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-compose-unit-4-pathway-1%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-compose-test-viewmodel#3)

Хорошая стратегия тестирования основана на охвате различных путей и границ вашего кода. На самом базовом уровне вы можете разделить тесты на три сценария: путь успеха, путь ошибки и граничный случай.

* **Путь успеха.** Тесты пути успеха, также известные как тесты счастливого пути, сосредоточены на тестировании функциональности для положительного потока. Положительный поток — это поток, который не имеет исключений или ошибок. По сравнению со сценариями пути ошибки и граничного случая легко создать исчерпывающий список сценариев путей успеха, поскольку они ориентированы на предполагаемое поведение вашего приложения.

Примером успеха в приложении Unscramble является правильное обновление счета, количества слов и зашифрованного слова, когда пользователь вводит правильное слово и нажимает кнопку **«Отправить»** .

* **Путь ошибки.** Тесты пути ошибки направлены на тестирование функциональности на предмет отрицательного потока, то есть на проверку того, как приложение реагирует на условия ошибки или неверный ввод пользователя. Определить все возможные потоки ошибок довольно сложно, поскольку существует множество возможных результатов, когда запланированное поведение не достигается.

Один общий совет — составить список всех возможных путей ошибок, написать для них тесты и постоянно развивать модульные тесты по мере обнаружения различных сценариев.

Примером ошибки в приложении Unscramble является то, что пользователь вводит неправильное слово и нажимает кнопку **«Отправить»** , в результате чего появляется сообщение об ошибке, а счет и количество слов не обновляются.

* **Граничный случай.** Граничный случай фокусируется на тестировании граничных условий в приложении. В приложении Unscramble граница проверяет состояние пользовательского интерфейса при загрузке приложения и состояние пользовательского интерфейса после того, как пользователь произнесет максимальное количество слов.

Создание сценариев тестирования на основе этих категорий может послужить руководством для вашего плана тестирования.

## Создание тестов

Хороший модульный тест обычно имеет следующие четыре свойства:

* **Сосредоточенность:** она должна быть сосредоточена на тестировании какого-либо модуля, например фрагмента кода. Этот фрагмент кода часто является классом или методом. Тест должен быть узким и фокусироваться на проверке правильности отдельных фрагментов кода, а не нескольких фрагментов кода одновременно.
* **Понятность:** код должен быть простым и понятным при чтении. С первого взгляда разработчик должен иметь возможность сразу понять цель теста.
* **Детерминированный:** он должен последовательно проходить или проваливаться. Когда вы запускаете тесты любое количество раз, не внося никаких изменений в код, тест должен давать тот же результат. Тест не должен быть нестабильным, с провалом в одном случае и прохождением в другом, несмотря на отсутствие изменений в коде.
* **Автономный:** он не требует какого-либо вмешательства человека или настройки и работает изолированно.

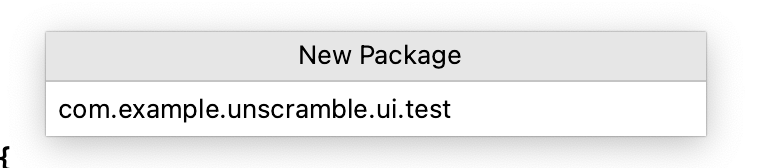
### **Путь успеха**

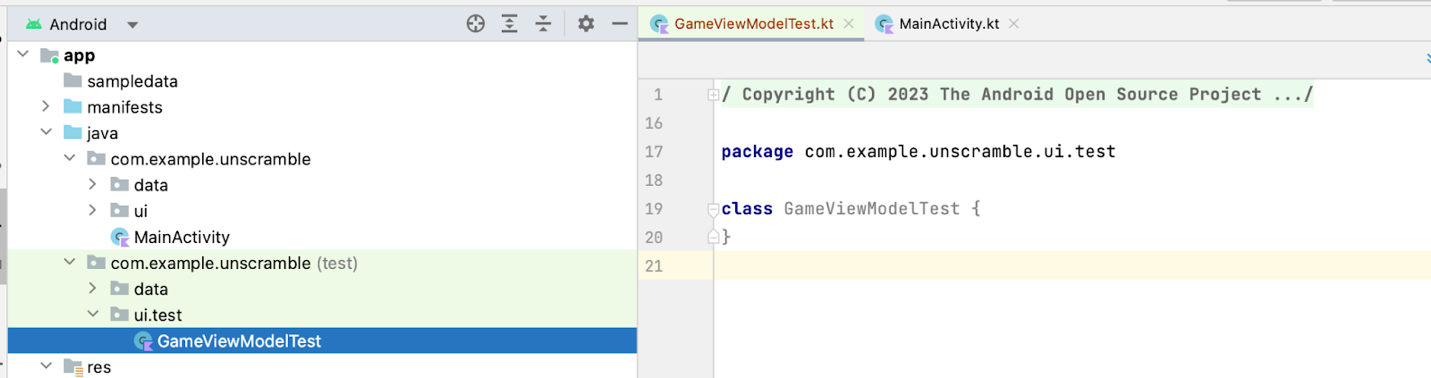
Чтобы написать модульный тест для пути успеха, вам необходимо утверждать, что, учитывая, что экземпляр GameViewModelбыл инициализирован, когда updateUserGuess()метод вызывается с правильным предполагаемым словом, за которым следует вызов checkUserGuess()метода, тогда:

* Правильное предположение передается методу updateUserGuess().
* Метод checkUserGuess()называется.
* Значение scoreи isGuessedWordWrongстатус обновляются правильно.

Для создания теста выполните следующие шаги:

1. Создайте новый пакет com.example.android.unscramble.ui.testв тестовом наборе исходных кодов и добавьте файл, как показано на следующем снимке экрана:





Чтобы написать модульный тест для GameViewModelкласса, вам понадобится экземпляр класса, чтобы вы могли вызывать методы класса и проверять состояние.

1. В теле класса GameViewModelTestобъявите свойство и присвойте ему viewModelэкземпляр класса .GameViewModel

class GameViewModelTest {  
    private val viewModel = GameViewModel()  
}

1. Чтобы написать модульный тест для успешного пути, создайте gameViewModel\_CorrectWordGuessed\_ScoreUpdatedAndErrorFlagUnset()функцию и добавьте к ней @Testаннотацию.

class GameViewModelTest {  
    private val viewModel = GameViewModel()  
  
    @Test  
    fun gameViewModel\_CorrectWordGuessed\_ScoreUpdatedAndErrorFlagUnset()  {  
    }  
}

1. Импортируйте следующее:

import org.junit.Test

**Примечание.** В приведенном выше коде используется thingUnderTest\_TriggerOfTest\_ResultOfTestформат имени тестовой функции:

* thingUnderTest"="gameViewModel
* TriggerOfTest"="CorrectWordGuessed
* ResultOfTest "="ScoreUpdatedAndErrorFlagUnset

Чтобы передать методу правильное слово игрока viewModel.updateUserGuess(), вам необходимо получить правильное расшифрованное слово из зашифрованного слова в формате GameUiState. Для этого сначала получите текущее состояние пользовательского интерфейса игры.

1. В теле функции создайте currentGameUiStateпеременную и присвойте viewModel.uiState.valueей значение.

@Test  
fun gameViewModel\_CorrectWordGuessed\_ScoreUpdatedAndErrorFlagUnset() {  
    var currentGameUiState = viewModel.uiState.value  
}

**Предупреждение** : этот способ получения uiState работает, поскольку вы использовали MutableStateFlow. В следующих модулях вы узнаете о расширенных способах использования, StateFlowкоторые создают поток данных, и вам нужно реагировать на обработку этого потока. Для этих сценариев вы будете писать модульные тесты, используя разные методы и подходы.

1. Чтобы получить правильное предположение игрока, используйте getUnscrambledWord()функцию, которая принимает в currentGameUiState.currentScrambledWordкачестве аргумента и возвращает расшифрованное слово. Сохраните это возвращаемое значение в новой переменной, доступной только для чтения, correctPlayerWordи присвойте ей значение, возвращаемое функцией getUnscrambledWord().

@Test  
fun gameViewModel\_CorrectWordGuessed\_ScoreUpdatedAndErrorFlagUnset() {  
    var currentGameUiState = viewModel.uiState.value  
    val correctPlayerWord = getUnscrambledWord(currentGameUiState.currentScrambledWord)  
  
}

1. Чтобы проверить правильность угаданного слова, добавьте вызов метода viewModel.updateUserGuess()и передайте correctPlayerWordпеременную в качестве аргумента. Затем добавьте вызов viewModel.checkUserGuess()метода, чтобы проверить предположение.

@Test  
fun gameViewModel\_CorrectWordGuessed\_ScoreUpdatedAndErrorFlagUnset() {  
    var currentGameUiState = viewModel.uiState.value  
    val correctPlayerWord = getUnscrambledWord(currentGameUiState.currentScrambledWord)  
  
    viewModel.updateUserGuess(correctPlayerWord)  
    viewModel.checkUserGuess()  
}

Теперь вы готовы утверждать, что состояние игры соответствует вашим ожиданиям.

1. Получите экземпляр класса GameUiStateиз значения свойства viewModel.uiStateи сохраните его в currentGameUiStateпеременной.

@Test  
fun gameViewModel\_CorrectWordGuessed\_ScoreUpdatedAndErrorFlagUnset() {  
    var currentGameUiState = viewModel.uiState.value  
    val correctPlayerWord = getUnscrambledWord(currentGameUiState.currentScrambledWord)  
    viewModel.updateUserGuess(correctPlayerWord)  
    viewModel.checkUserGuess()  
  
    currentGameUiState = viewModel.uiState.value  
}

1. Чтобы проверить правильность угаданного слова и обновить оценку, используйте функцию assertFalse()для проверки того, что currentGameUiState.isGuessedWordWrongсвойство имеет значение, falseи assertEquals()функцию для проверки того, что значение свойства currentGameUiState.scoreравно 20.

@Test  
fun gameViewModel\_CorrectWordGuessed\_ScoreUpdatedAndErrorFlagUnset() {  
    var currentGameUiState = viewModel.uiState.value  
    val correctPlayerWord = getUnscrambledWord(currentGameUiState.currentScrambledWord)  
    viewModel.updateUserGuess(correctPlayerWord)  
    viewModel.checkUserGuess()  
  
    currentGameUiState = viewModel.uiState.value  
    // Assert that checkUserGuess() method updates isGuessedWordWrong is updated correctly.  
    assertFalse(currentGameUiState.isGuessedWordWrong)  
    // Assert that score is updated correctly.  
    assertEquals(20, currentGameUiState.score)  
}

1. Импортируйте следующее:

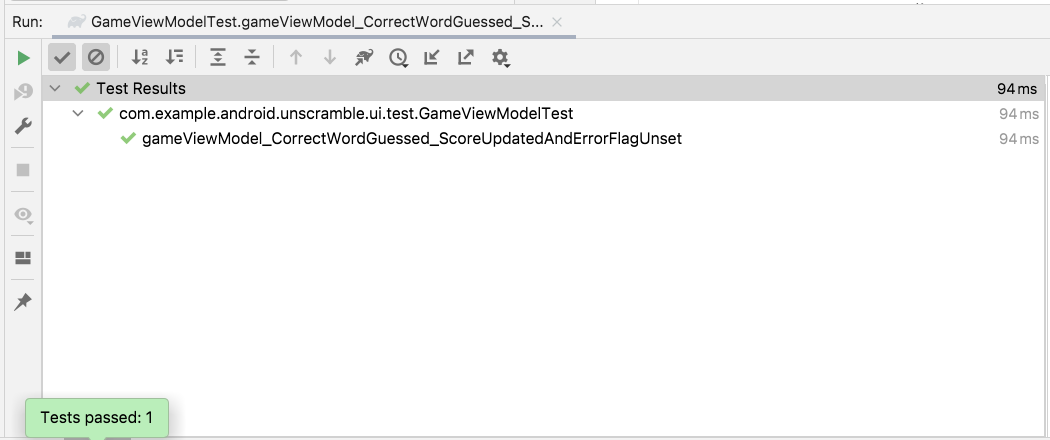
import org.junit.Assert.assertEquals  
import org.junit.Assert.assertFalse

1. Чтобы сделать значение 20доступным для чтения и повторного использования, создайте сопутствующий объект и присвойте 20ему privateконстанту с именем SCORE\_AFTER\_FIRST\_CORRECT\_ANSWER. Обновите тест, используя вновь созданную константу.

class GameViewModelTest {  
    ...  
    @Test  
    fun gameViewModel\_CorrectWordGuessed\_ScoreUpdatedAndErrorFlagUnset() {  
        ...  
        // Assert that score is updated correctly.  
        assertEquals(SCORE\_AFTER\_FIRST\_CORRECT\_ANSWER, currentGameUiState.score)  
    }  
  
    companion object {  
        private const val SCORE\_AFTER\_FIRST\_CORRECT\_ANSWER = SCORE\_INCREASE  
    }  
}

1. Запустите тест.

Тест должен пройти успешно, поскольку все утверждения верны, как показано на следующем снимке экрана:



### **Путь ошибки**

Чтобы написать модульный тест для пути ошибки, вам нужно утверждать, что когда в качестве аргумента метода передается неверное слово viewModel.updateUserGuess()и viewModel.checkUserGuess()метод вызывается, происходит следующее:

* Стоимость недвижимости currentGameUiState.scoreостается неизменной.
* Значение свойства currentGameUiState.isGuessedWordWrongустановлено, trueпоскольку предположение неверно.

Для создания теста выполните следующие шаги:

1. В теле класса GameViewModelTestсоздайте gameViewModel\_IncorrectGuess\_ErrorFlagSet()функцию и добавьте к ней аннотацию @Test.

@Test  
fun gameViewModel\_IncorrectGuess\_ErrorFlagSet() {  
      
}

1. Определите incorrectPlayerWordпеременную и присвойте "and"ей значение, которого не должно быть в списке слов.

@Test  
fun gameViewModel\_IncorrectGuess\_ErrorFlagSet() {  
    // Given an incorrect word as input  
    val incorrectPlayerWord = "and"  
}

1. Добавьте вызов метода viewModel.updateUserGuess()и передайте incorrectPlayerWordпеременную в качестве аргумента.
2. Добавьте вызов viewModel.checkUserGuess()метода для проверки предположения.

@Test  
fun gameViewModel\_IncorrectGuess\_ErrorFlagSet() {  
    // Given an incorrect word as input  
    val incorrectPlayerWord = "and"  
  
    viewModel.updateUserGuess(incorrectPlayerWord)  
    viewModel.checkUserGuess()  
}

1. Добавьте переменную и присвойте ей currentGameUiStateзначение состояния .viewModel.uiState.value
2. Используйте функции утверждения, чтобы подтвердить, что значение свойства currentGameUiState.scoreравно 0и значение currentGameUiState.isGuessedWordWrongсвойства установлено на true.

@Test  
fun gameViewModel\_IncorrectGuess\_ErrorFlagSet() {  
    // Given an incorrect word as input  
    val incorrectPlayerWord = "and"  
  
    viewModel.updateUserGuess(incorrectPlayerWord)  
    viewModel.checkUserGuess()  
  
    val currentGameUiState = viewModel.uiState.value  
    // Assert that score is unchanged  
    assertEquals(0, currentGameUiState.score)  
    // Assert that checkUserGuess() method updates isGuessedWordWrong correctly  
    assertTrue(currentGameUiState.isGuessedWordWrong)  
}

1. Импортируйте следующее:

import org.junit.Assert.assertTrue

1. Запустите тест, чтобы убедиться, что он пройден.

### **Граничный случай**

Чтобы протестировать исходное состояние пользовательского интерфейса, вам нужно написать модульный тест для GameViewModelкласса. Тест должен утверждать, что при GameViewModelинициализации выполняется следующее:

* currentWordCountсвойство установлено на 1.
* scoreсвойство установлено на 0.
* isGuessedWordWrongсвойство установлено на false.
* isGameOverсвойство установлено на false.

Чтобы добавить тест, выполните следующие действия:

1. Создайте gameViewModel\_Initialization\_FirstWordLoaded()метод и добавьте к нему @Testаннотацию.

@Test  
fun gameViewModel\_Initialization\_FirstWordLoaded() {  
      
}

1. Доступ к viewModel.uiState.valueсвойству для получения исходного экземпляра класса GameUiState. Назначьте его новой gameUiStateпеременной, доступной только для чтения.

@Test  
fun gameViewModel\_Initialization\_FirstWordLoaded() {  
    val gameUiState = viewModel.uiState.value  
}

1. Чтобы получить правильное слово игрока, используйте getUnscrambledWord()функцию, которая принимает gameUiState.currentScrambledWordслово и возвращает расшифрованное слово. Присвойте возвращаемое значение новой переменной, доступной только для чтения, с именем unScrambledWord.

@Test  
fun gameViewModel\_Initialization\_FirstWordLoaded() {  
    val gameUiState = viewModel.uiState.value  
    val unScrambledWord = getUnscrambledWord(gameUiState.currentScrambledWord)  
  
}

1. Чтобы проверить правильность состояния, добавьте assertTrue()функции, проверяющие, что для currentWordCountсвойства установлено значение 1, а для scoreсвойства установлено значение 0.
2. Добавьте assertFalse()функции, чтобы убедиться, что свойство isGuessedWordWrongимеет значение .falseisGameOverfalse

@Test  
fun gameViewModel\_Initialization\_FirstWordLoaded() {  
    val gameUiState = viewModel.uiState.value  
    val unScrambledWord = getUnscrambledWord(gameUiState.currentScrambledWord)  
  
    // Assert that current word is scrambled.  
    assertNotEquals(unScrambledWord, gameUiState.currentScrambledWord)  
    // Assert that current word count is set to 1.  
    assertTrue(gameUiState.currentWordCount == 1)  
    // Assert that initially the score is 0.  
    assertTrue(gameUiState.score == 0)  
    // Assert that the wrong word guessed is false.  
    assertFalse(gameUiState.isGuessedWordWrong)  
    // Assert that game is not over.  
    assertFalse(gameUiState.isGameOver)  
}

1. Импортируйте следующее:

import org.junit.Assert.assertNotEquals

1. Запустите тест, чтобы убедиться, что он пройден.

Другой пограничный случай — проверка состояния пользовательского интерфейса после того, как пользователь угадает все слова. Вам нужно утверждать, что когда пользователь правильно угадывает все слова, верно следующее:

* Оценка актуальна;
* Свойство currentGameUiState.currentWordCountравно значению константы MAX\_NO\_OF\_WORDS;
* Свойство currentGameUiState.isGameOverустановлено на true.

Чтобы добавить тест, выполните следующие действия:

1. Создайте gameViewModel\_AllWordsGuessed\_UiStateUpdatedCorrectly()метод и добавьте к нему @Testаннотацию. В методе создайте expectedScoreпеременную и присвойте 0ей значение.

@Test  
fun gameViewModel\_AllWordsGuessed\_UiStateUpdatedCorrectly() {  
    var expectedScore = 0  
}

1. Чтобы получить исходное состояние, добавьте currentGameUiStateпеременную и присвойте ей значение свойства viewModel.uiState.value.

@Test  
fun gameViewModel\_AllWordsGuessed\_UiStateUpdatedCorrectly() {  
    var expectedScore = 0  
    var currentGameUiState = viewModel.uiState.value  
}

1. Чтобы получить правильное слово игрока, используйте getUnscrambledWord()функцию, которая принимает currentGameUiState.currentScrambledWordслово и возвращает расшифрованное слово. Сохраните это возвращаемое значение в новой переменной, доступной только для чтения, correctPlayerWordи присвойте ей значение, возвращаемое функцией getUnscrambledWord().

@Test  
fun gameViewModel\_AllWordsGuessed\_UiStateUpdatedCorrectly() {  
    var expectedScore = 0  
    var currentGameUiState = viewModel.uiState.value  
    var correctPlayerWord = getUnscrambledWord(currentGameUiState.currentScrambledWord)  
}

1. Чтобы проверить, угадает ли пользователь все ответы, используйте repeatблок, повторяющий выполнение метода viewModel.updateUserGuess()и время выполнения viewModel.checkUserGuess()метода MAX\_NO\_OF\_WORDS.

@Test  
fun gameViewModel\_AllWordsGuessed\_UiStateUpdatedCorrectly() {  
    var expectedScore = 0  
    var currentGameUiState = viewModel.uiState.value  
    var correctPlayerWord = getUnscrambledWord(currentGameUiState.currentScrambledWord)  
    repeat(MAX\_NO\_OF\_WORDS) {  
          
    }  
}

1. В repeatблоке добавьте значение константы SCORE\_INCREASEк expectedScoreпеременной, чтобы утверждать, что оценка увеличивается после каждого правильного ответа.
2. Добавьте вызов метода viewModel.updateUserGuess()и передайте correctPlayerWordпеременную в качестве аргумента.
3. Добавьте вызов метода viewModel.checkUserGuess(), чтобы инициировать проверку предположения пользователя.

@Test  
fun gameViewModel\_AllWordsGuessed\_UiStateUpdatedCorrectly() {  
    var expectedScore = 0  
    var currentGameUiState = viewModel.uiState.value  
    var correctPlayerWord = getUnscrambledWord(currentGameUiState.currentScrambledWord)  
    repeat(MAX\_NO\_OF\_WORDS) {  
        expectedScore += SCORE\_INCREASE  
        viewModel.updateUserGuess(correctPlayerWord)  
        viewModel.checkUserGuess()  
    }  
}

1. Обновите текущее слово игрока, используйте getUnscrambledWord()функцию, которая принимает в currentGameUiState.currentScrambledWordкачестве аргумента и возвращает расшифрованное слово. Сохраните это возвращенное значение в новой переменной, доступной только для чтения, с именем. correctPlayerWord.Чтобы проверить правильность состояния, добавьте assertEquals()функцию, проверяющую, currentGameUiState.scoreравно ли значение свойства значению expectedScoreпеременной.

@Test  
fun gameViewModel\_AllWordsGuessed\_UiStateUpdatedCorrectly() {  
    var expectedScore = 0  
    var currentGameUiState = viewModel.uiState.value  
    var correctPlayerWord = getUnscrambledWord(currentGameUiState.currentScrambledWord)  
    repeat(MAX\_NO\_OF\_WORDS) {  
        expectedScore += SCORE\_INCREASE  
        viewModel.updateUserGuess(correctPlayerWord)  
        viewModel.checkUserGuess()  
        currentGameUiState = viewModel.uiState.value  
        correctPlayerWord = getUnscrambledWord(currentGameUiState.currentScrambledWord)  
        // Assert that after each correct answer, score is updated correctly.  
        assertEquals(expectedScore, currentGameUiState.score)  
    }  
}

1. Добавьте assertEquals()функцию, которая будет утверждать, что значение свойства currentGameUiState.currentWordCountравно значению константы MAX\_NO\_OF\_WORDSи что значение свойства currentGameUiState.isGameOverустановлено равным true.

@Test  
fun gameViewModel\_AllWordsGuessed\_UiStateUpdatedCorrectly() {  
    var expectedScore = 0  
    var currentGameUiState = viewModel.uiState.value  
    var correctPlayerWord = getUnscrambledWord(currentGameUiState.currentScrambledWord)  
    repeat(MAX\_NO\_OF\_WORDS) {  
        expectedScore += SCORE\_INCREASE  
        viewModel.updateUserGuess(correctPlayerWord)  
        viewModel.checkUserGuess()  
        currentGameUiState = viewModel.uiState.value  
        correctPlayerWord = getUnscrambledWord(currentGameUiState.currentScrambledWord)  
        // Assert that after each correct answer, score is updated correctly.  
        assertEquals(expectedScore, currentGameUiState.score)  
    }  
    // Assert that after all questions are answered, the current word count is up-to-date.  
    assertEquals(MAX\_NO\_OF\_WORDS, currentGameUiState.currentWordCount)  
    // Assert that after 10 questions are answered, the game is over.  
    assertTrue(currentGameUiState.isGameOver)  
}

1. Импортируйте следующее:

import com.example.unscramble.data.MAX\_NO\_OF\_WORDS

1. Запустите тест, чтобы убедиться, что он пройден.

## ****Обзор жизненного цикла тестового экземпляра****

Если вы внимательно посмотрите на способ viewModelинициализации в тесте, вы можете заметить, что он viewModelинициализируется только один раз, хотя все тесты его используют. Этот фрагмент кода показывает определение свойства viewModel.

class GameViewModelTest {  
    private val viewModel = GameViewModel()  
      
    @Test  
    fun gameViewModel\_Initialization\_FirstWordLoaded() {  
        val gameUiState = viewModel.uiState.value  
        ...  
    }  
    ...  
}

У вас могут возникнуть следующие вопросы:

* Означает ли это, что один и тот же экземпляр viewModelповторно используется для всех тестов?
* Вызовет ли это какие-либо проблемы? Например, что, если gameViewModel\_Initialization\_FirstWordLoadedтестовый метод выполняется после gameViewModel\_CorrectWordGuessed\_ScoreUpdatedAndErrorFlagUnsetтестового метода? Не пройдет ли тест инициализации?

Ответ на оба вопроса – нет. Методы тестирования выполняются изолированно, чтобы избежать неожиданных побочных эффектов из изменяемого состояния экземпляра теста. По умолчанию перед выполнением каждого тестового метода JUnit создает новый экземпляр тестового класса.

GameViewModelTestПоскольку в вашем классе уже есть четыре метода тестирования , они GameViewModelTestсоздаются четыре раза. Каждый экземпляр имеет свою собственную копию свойства viewModel. Следовательно, последовательность выполнения тестов не имеет значения.

**Примечание.** Этот жизненный цикл экземпляра теста «для каждого метода» является поведением по умолчанию, начиная с JUnit4.

[5. Введение в покрытие кода](https://developer.android.com/codelabs/basic-android-kotlin-compose-test-viewmodel?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-compose-unit-4-pathway-1%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-compose-test-viewmodel#4)

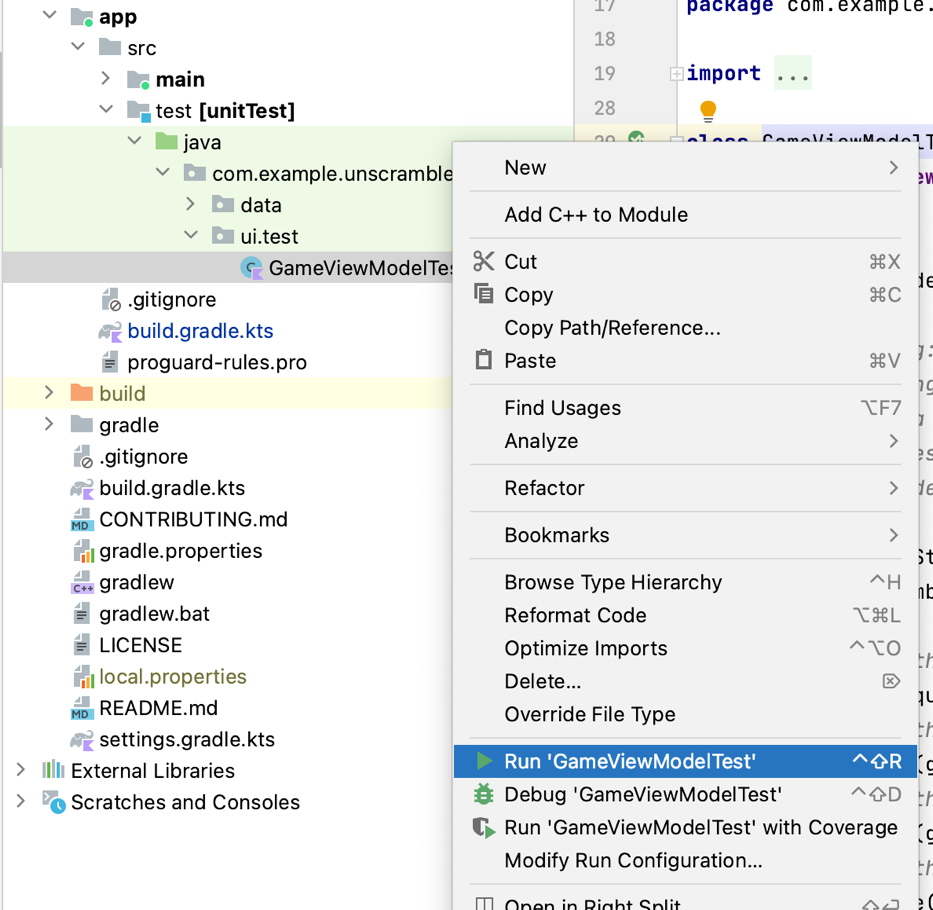
Покрытие кода играет жизненно важную роль, определяя, насколько адекватно вы тестируете классы, методы и строки кода, составляющие ваше приложение.

Android Studio предоставляет инструмент тестового покрытия для локальных модульных тестов, позволяющий отслеживать процент и области кода вашего приложения, охваченные вашими модульными тестами.

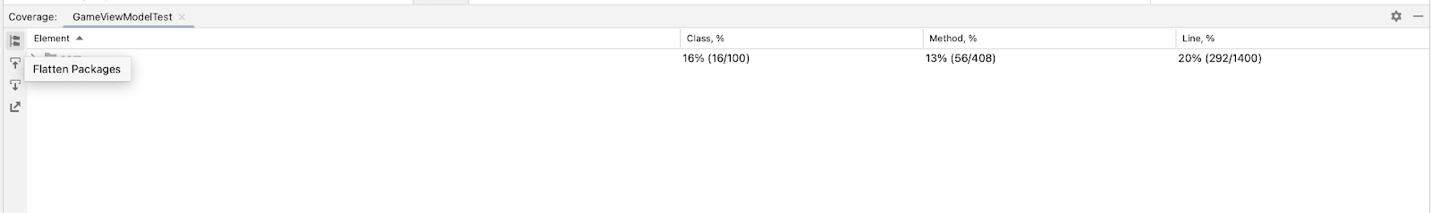
**Запустите тест с покрытием с помощью Android Studio.**

Чтобы запустить тесты с покрытием:

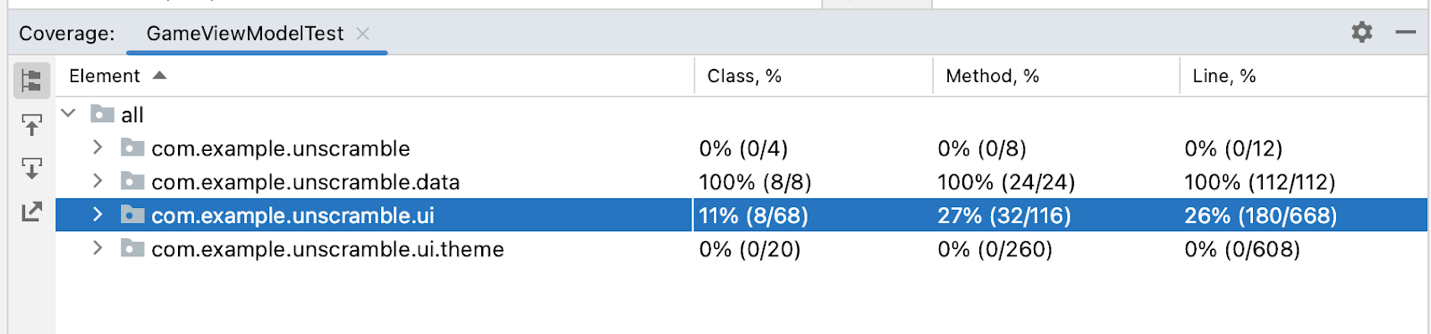
1. Щелкните правой кнопкой мыши GameViewModelTest.ktфайл на панели проекта и выберите  **«Запустить GameViewModelTest» с покрытием** .



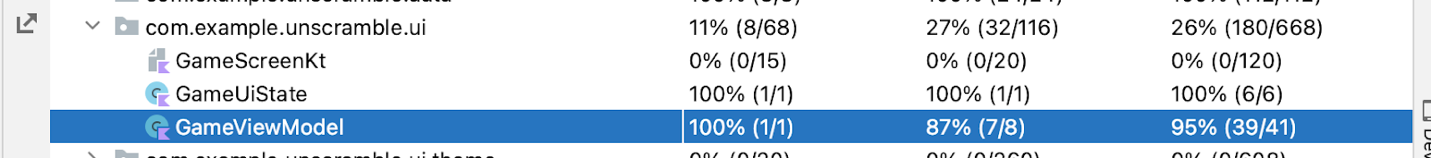
1. После завершения выполнения теста на панели покрытия справа выберите параметр **«Свести пакеты»** .



1. Обратите внимание на com.example.android.unscramble.uiпакет, как показано на следующем изображении.



1. Дважды щелкните com.example.android.unscramble.ui имя пакета, чтобы отобразить покрытие, GameViewModelкак показано на следующем рисунке:



**Анализ отчета об испытаниях**

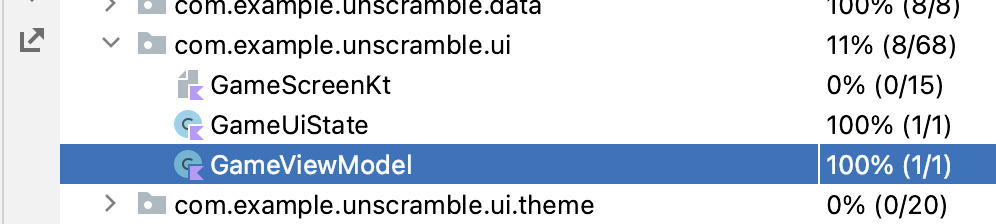
Отчет, показанный на следующей диаграмме, разбит на два аспекта:

* **Процент методов, охваченных модульными тестами.** На диаграмме примера написанные вами тесты на данный момент охватывают 7 из 8 методов. Это 87% от общего числа методов.
* **Процент строк, охваченных модульными тестами.** На диаграмме-примере написанные вами тесты охватывают 39 из 41 строки кода. Это 95% строк кода.

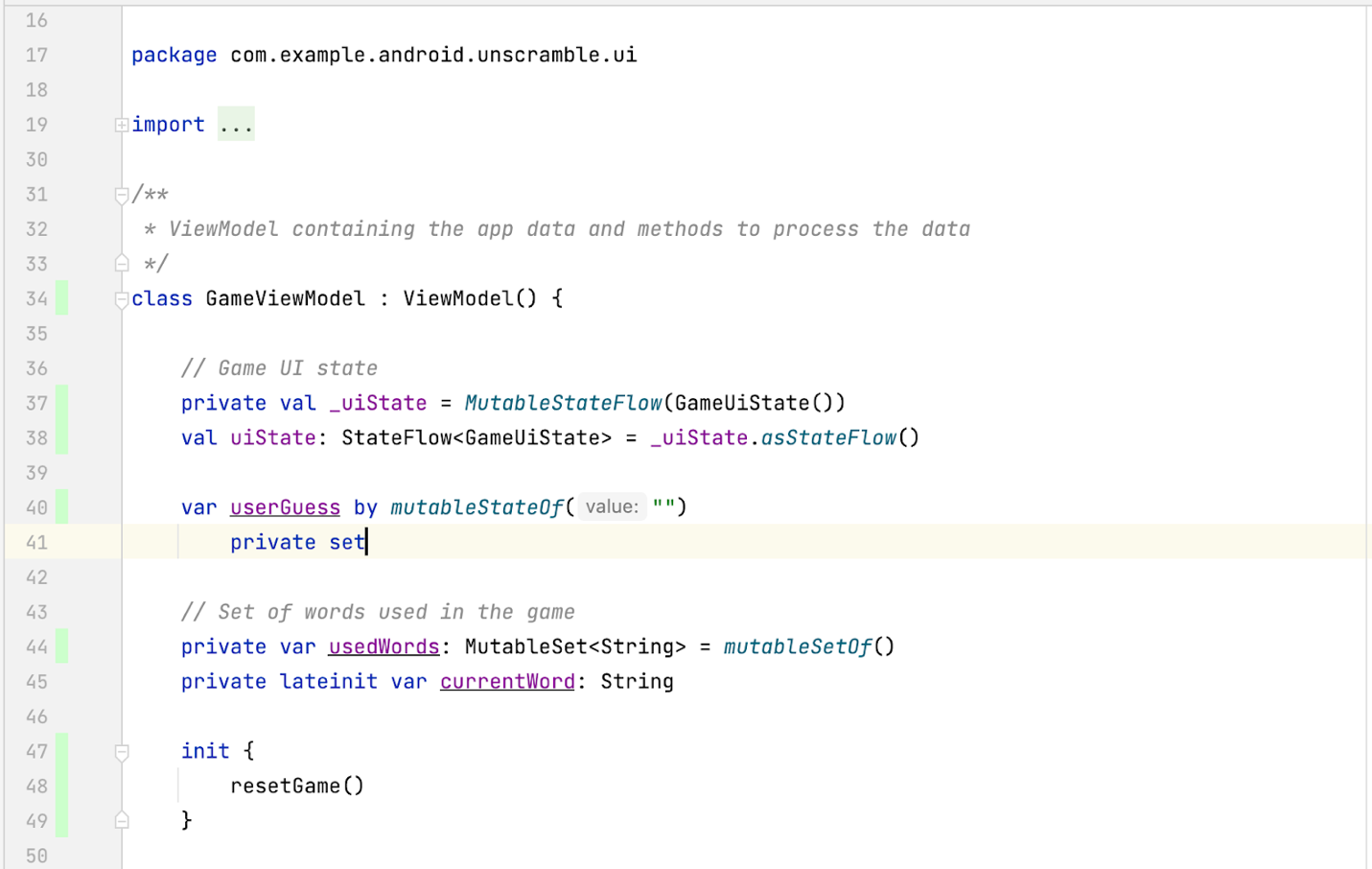
**Примечание.** Покрытие кода отслеживает количество строк, выполненных во время выполнения теста. Следовательно, даже несмотря на отсутствие проверок на GameUiState, покрытие показывает 100%.

Отчеты показывают, что написанные вами модульные тесты пропускают определенные части кода. Чтобы определить, какие части были пропущены, выполните следующий шаг:

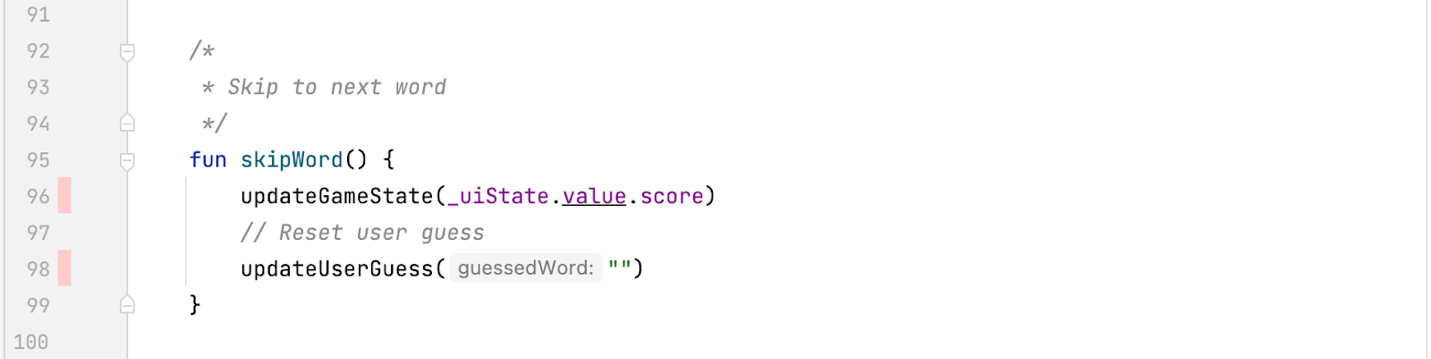
* Дважды щелкните **GameViewModel** .



Android Studio отображает GameViewModel.ktфайл с дополнительной цветовой кодировкой в ​​левой части окна. Ярко-зеленый цвет указывает на то, что эти строки кода были закрыты.



Прокрутив страницу вниз GameViewModel, вы можете заметить, что несколько строк отмечены светло-розовым цветом. Этот цвет указывает на то, что эти строки кода не были охвачены модульными тестами.



**Улучшить покрытие**

Чтобы улучшить покрытие, вам нужно написать тест, покрывающий недостающий путь. Вам нужно добавить тест, который будет утверждать, что когда пользователь пропускает слово, верно следующее:

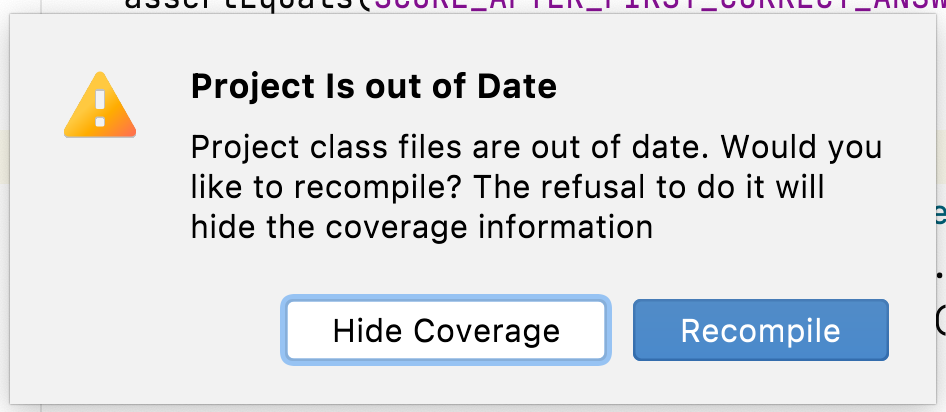
* currentGameUiState.scoreимущество остается неизменным.
* currentGameUiState.currentWordCountСвойство увеличивается на единицу, как показано в следующем фрагменте кода.

Чтобы подготовиться к улучшению покрытия, добавьте в класс следующий метод тестирования GameViewModelTest.

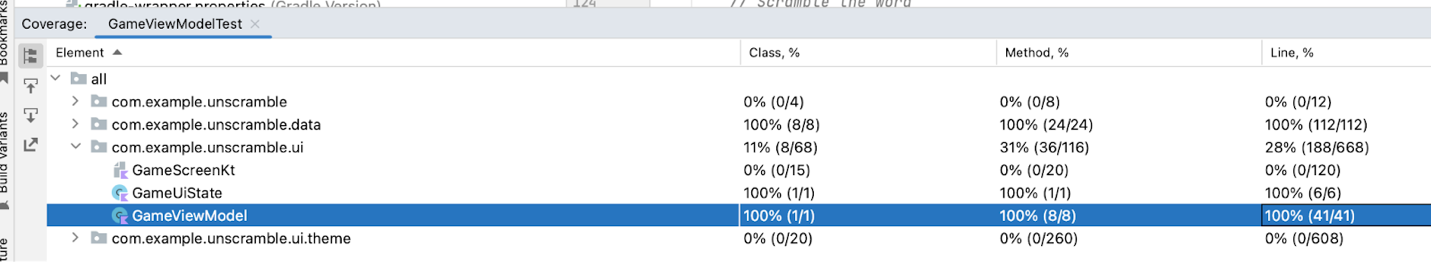
@Test  
fun gameViewModel\_WordSkipped\_ScoreUnchangedAndWordCountIncreased() {  
    var currentGameUiState = viewModel.uiState.value  
    val correctPlayerWord = getUnscrambledWord(currentGameUiState.currentScrambledWord)  
    viewModel.updateUserGuess(correctPlayerWord)  
    viewModel.checkUserGuess()  
  
    currentGameUiState = viewModel.uiState.value  
    val lastWordCount = currentGameUiState.currentWordCount  
    viewModel.skipWord()  
    currentGameUiState = viewModel.uiState.value  
    // Assert that score remains unchanged after word is skipped.  
    assertEquals(SCORE\_AFTER\_FIRST\_CORRECT\_ANSWER, currentGameUiState.score)  
    // Assert that word count is increased by 1 after word is skipped.  
    assertEquals(lastWordCount + 1, currentGameUiState.currentWordCount)  
}

Чтобы повторно запустить покрытие, выполните следующие действия:

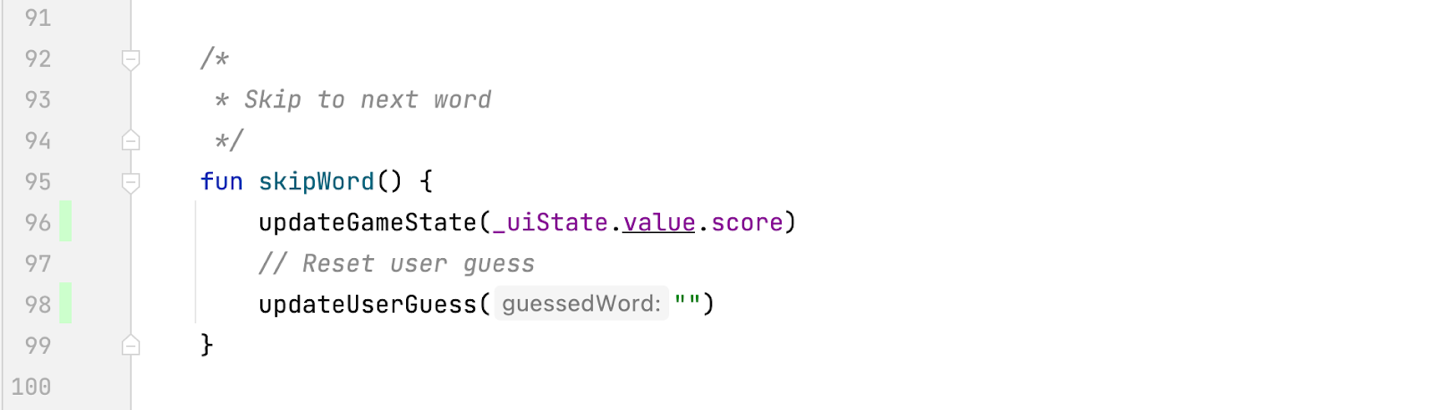
1. Щелкните GameViewModelTest.ktфайл правой кнопкой мыши и в меню выберите  **«Запустить GameViewModelTest» с покрытием** .
2. Чтобы перекомпилировать результаты после повторного запуска, нажмите кнопку **«Перекомпилировать»** , когда увидите приглашение, как показано на следующем рисунке:



1. После успешной сборки снова перейдите к элементу **GameViewModel** и убедитесь, что процент покрытия составляет 100%. Окончательный отчет о покрытии показан на следующем изображении.



1. Перейдите к GameViewModel.ktфайлу и прокрутите вниз, чтобы проверить, покрыт ли ранее пропущенный путь.



Вы узнали, как запускать, анализировать и улучшать покрытие кода вашего приложения.

*Означает ли высокий процент покрытия кода высокое качество кода приложения?* Нет. Покрытие кода показывает процент кода, покрытого или выполненного вашим модульным тестом. Это не означает, что код проверен. Если вы удалите все утверждения из кода модульного теста и запустите покрытие кода, покрытие по-прежнему будет 100%.

Высокое покрытие не означает, что тесты разработаны правильно и что тесты проверяют поведение приложения. Вам необходимо убедиться, что написанные вами тесты содержат утверждения, проверяющие поведение тестируемого класса. Вам также не нужно стремиться писать модульные тесты, чтобы получить 100% тестовое покрытие всего приложения. Вместо этого вам следует протестировать некоторые части кода приложения, например действия, используя тесты пользовательского интерфейса.

Однако низкое покрытие означает, что большие части вашего кода были совершенно непроверены. Используйте покрытие кода как инструмент для поиска частей кода, которые не были выполнены вашими тестами, а не как инструмент для измерения качества вашего кода.

## [7. Заключение](https://developer.android.com/codelabs/basic-android-kotlin-compose-test-viewmodel?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-compose-unit-4-pathway-1%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-compose-test-viewmodel#6)

Поздравляем! Вы узнали, как определить стратегию тестирования и внедрить модульные тесты для тестирования ViewModelи StateFlowв приложении Unscramble. Продолжая создавать приложения для Android, убедитесь, что вы пишете тесты вместе с функциями вашего приложения, чтобы убедиться, что ваши приложения работают правильно на протяжении всего процесса разработки.

### **Краткое содержание**

* Используйте testImplementationконфигурацию, чтобы указать, что зависимости применяются к локальному исходному коду теста, а не к коду приложения.
* Постарайтесь классифицировать тесты по трем сценариям: путь успеха, путь ошибки и граничный случай.
* Хороший модульный тест имеет как минимум четыре характеристики: он целенаправленный, понятный, детерминированный и автономный.
* Методы тестирования выполняются изолированно, чтобы избежать неожиданных побочных эффектов из изменяемого состояния экземпляра теста.
* По умолчанию перед выполнением каждого метода тестирования JUnit создает новый экземпляр тестового класса.
* Покрытие кода играет жизненно важную роль в определении того, адекватно ли вы протестировали классы, методы и строки кода, составляющие ваше приложение.

### **Узнать больше**

* [Основы тестирования Android-приложений](https://developer.android.com/training/testing/fundamentals#benefits)
* [Использование спецификации | Создание реактивного ранца | Android-разработчики](https://developer.android.com/jetpack/compose/bom/bom)