**Автоматическое тестирование** – проверка кода на его соответствие ожидаемому поведению в рамках сценария взаимодействия с данным кодом.

1. Фиксация ожидаемого поведения в виде теста
2. Возможность простого регрессионного тестирования - в случае добавления нового функционала дает возможность убедиться, что ранее написанный код не конфликтует с новым функционалом.

**Виды тестирования (с точки зрения функциональности):**

Модульное тестирование - тестирование отдельной сущности в изоляции от других компонентов (JUnit).

Интеграционное тестирование – тестирование взаимодействия между компонентами системы.

Функциональное тестирование – тестирование соответствия работы программы с бизнес требованиями, исследуя конечных результат взаимодействия без учета промежуточных состояний.

Acceptance – тестирование всего приложения со стороны пользователя (jbehave)

**Junit** – фреймворк для модульного тестирования, состоит из нескольких модулей: Platform – движок для запуска тестов на JVM, Jupiter – классы для разработки и написания тестов (API для написания тестов), vintage – для обратной совместимости с предыдущими версиями Junit.

*Общий вид написания тестов*:

1. Именование теста через Camel Case или Snake case, название должно точно отражать, какой сценарий тестирует данный тест.
2. Не должно быть зависимостей между тестами (порядок выполнения тестов не должен влиять на их результат).
3. Каждый тест проверяет именно сущность, а не работу классов, используемых в ней. Тестируется один сценарий, и его результат не должен зависеть от результатов другого теста.
4. Каждый сценарий проверяется на предмет соответствия ожидаемого значения с получаемым значением от выполнения функции, их соответствие проверяется методами типа assert:

|  |  |
| --- | --- |
| assertAll(Executable…ex) | Принимает множество Executable (не принимает и не возвращает ничего), позволяет выполнять несколько assert внутри одного теста, они будут выполнены даже если один из них не пройдет. При написании в одном тестовом методе нескольких assert, если один не проходит, то следующие не исполняются. |
| assertEquals/assertNotEquals | Проверка на эквивалентность. При проверке примитивов сравнивается значение, для объектов по equals |
| assertArrayEquals | Проверяет равенство массивов по содержимому |
| assertTrue/assertFalse | Проверка на истинность логического утверждения |
| assertThrows(Class expected type, Executable ex) | Первым параметром принимает класс ожидаемого исключения, второй параметр – executable, который исполнит проверяемый метод |
| assertNull/assertNotNull | Проверка содержимое на соответствие null или not null |
| assertSame/assertNotSame | Эквивалентность через == |
| fail | Принудительно роняет тест |

**Жизненный цикл тестов** – порядок исполнения тестов, который определяется аннотациями:

@BeforeAll – метод будет исполнен перед всеми методами

@BeforeEach – метод будет исполнен перед каждым запуском @Test

@Test – тестовый метод

@AfterEach – Метод будет исполнен после каждого @Test

@AfterAll - Метод будет исполнен после всех методов

\*BeforeAll и AfterAll должны быть static при Lifecycle.Per\_Method (в этом контексте для каждого теста создается новый объект, а BeforeAll и AfterAll предназначены для выполнения один раз перед всеми тестами в классе, и статический контекст гарантирует, что это произойдет до создания экземпляров тестового класса, при TestInstance.Lifecycke.Per\_Class такой гарантии не требуется, т.к все равно создается только один объект для всех тестов).

*@TestInstance* – позволяет настроить жизненный цикл тестов, имеет два параметра TestInstance.Lifecycke.Per\_Class и TestInstance.Lifecycle.Per\_Method, которые определяют кол-во созданных объектов класса для каждого тестового метода.

Per\_Method – используется по умолчанию, создает экземпляр тестового класса для каждого метода @Test

Per\_Class – использует один объект тестового класса, используется в случае, если дорогая инициализация экземпляра класса, намеренное использование разделяемого состояния, т.е. возможно изменение состояния объекта в одном тесте и использование его уже в другом).

@Tag – Помечает тестовый класс или метод пользовательскими метками, что позволяет фильтровать тесты и запускать только необходимые, в зависимости от их тэга. @Tags – для более удобного совмещения нескольких тэгов над одним методом, классом.

mvn clean test -Dgroups=ТЭГ – запуск тестов определенного тэга.

mvn clean test -DexcludedGroups=ТЭГ – запускает все тесты, кроме этого тэга.

@Disabled – тест будет пропущен.

@RepeatedTest – тест будет запущен указанное кол-во раз (для проверки на стабильность теста)

@TimeOut – для установки временного ограничения выполнения теста. Есть ассерты

**Порядок исполнения тестов** - при запуске не определен, для задания порядка используется аннотация @TestMethodOrder.

@TestMethodOrder – определяет порядок выполнения тестов, для того, чтобы задать порядок, нужно передать в параметр один из классов наследников MethodOrder:

MethodName – по названию методов в алфавитном порядке

DisplayName – по заданному имени в алфавитном порядке, если оно не задано, то используется название метода.

OrderAnnotation – каждый тест помечается @Order, который в качестве параметра принимает порядок исполнения теста.

Random – рандомное исполнение тестов

@DisplayName – задает имя, которое будет использоваться во время отчета о выполнении тестов.

@Nested – для обозначения вложенных тестовых классов, используется для логической группировки тестов и для возможной изоляции этих логически связанных частей.

**Extension Model** - модель, предоставляющая возможность расширять функциональность тестов и внедрять зависимости с помощью расширений.

Test Instance Post-Processing — возможность модифицировать или настраивать экземпляр тестового класса после его создания.

Test Lifecycle Callbacks — дает возможность расширять жизненный цикл тестов (внедрять новые фазы в ЖЦ тестом) путем имплементации интерфейсов BeforeAllCallback, AfterAllCallback, BeforeEachCallback, и AfterEachCallback и др.

Parameter Resolution — внедрение параметров в методы или конструкторы тестов с помощью ParameterResolver. Это позволяет динамически передавать зависимости на основе типа параметра.

Exception Handling — управление исключениями, возникающими в тестах, с помощью TestExecutionExceptionHandler. Это дает возможность обрабатывать или перехватывать ошибки, чтобы выполнять определенные действия при их возникновении.

Conditional Test Execution — условное выполнение тестов с использованием ExecutionCondition. Например, можно отключить тест в зависимости от состояния окружения или конфигурации.

Для регистрации расширений, которые реализуют конкретные возможности модели расширений используется аннотация @ExtendedWith.

**Параметризованные тесты**

Для использования нужно подключить зависимость junit-jupiter-params

@ParameterizedTest – используется для запуска теста с несколькими параметрами. Требует определить источник аргументов с помощью @ArgumentSource, который в качестве параметра требует класс, который реализует интерфейс ArgumentProvider, который и будет определять источник параметров (если нужно реализовать свой) или использовать готовые аннотации, которые определяют источник параметров через уже готовые реализации ArgumentProvider.

@NullSource – предоставляет параметр null

@EmprySource - предоставляет пустой объект, определен только для некоторых классов (String, Lies, Set, Map, arrays)

@NullAndEmptySource – предоставляет два предыдущих аргумента

@ValueSource – предоставляет набор заданных значений (определены в документации)

@EnumSource – тестовый метод будет вызываться для каждой константы в переданном enum классе.

@MethodSource – использует метод-источник (передается в аннотацию в качестве пути к этому методу), который должен быть static и возвращать Stream<Argument>, если будет передаваться несколько аргументов или любой другой Stream, если будет передаваться один аргумент.

Пример:

//Тестируется метод, который получает строку и возвращает ее длину,   
// null или пустая строка возвращает 0  
 @ParameterizedTest  
 @MethodSource("org.example.MainTest#newMethod") //путь к источнику аргументов  
 void tester(Integer val, String line) {  
 *assertEquals*(val, Main.*length*(line));   
 }  
 static Stream<Arguments> newMethod() {  
 return Stream.*of*( //т.к аргумента два, то нужно заключить их в Arguments  
 Arguments.*of*(2, "ss"),  
 Arguments.*of*(1, "s"),   
 Arguments.*of*(5, "sssss"),  
 Arguments.*of*(0, null),  
 Arguments.*of*(0, "")  
 );  
 }

@CsvSource – определяет параметры непосредственно в аннотации, кол-во вызов будет соответствовать кол-ву Csv строк в токене, Csv-аннотации настраиваются с помощью параметров типа value – массив строк, которые будут использоваться в качестве параметров, delimiter – разделитель параметров, numEmptyLines – кол-во строк, которые нужно пропустить и т.д. (смотри доку).

@ParameterizedTest  
@CsvSource(value = {  
 "Первый параметр, Второй параметр",  
 "Первый параметр, Второй параметр"  
})  
void tester(String l1, String l2)

@CsvFileSource – практически тоже самое, только в качестве источника параметров используется Csv файл.

Минус использования Csv аннотаций – невозможность передать объекты в качестве параметра.

@ArgumentsSource – в качестве источника передается класс-наследник ArgumentProvider, который реализует метод, содержащий источник параметров (по сути такой же как в @MethodSource)

**AssertJ, Hamcrest** – библиотеки, которые предоставляют более обширный API для написания тестов (более читаемые).

**TDD** – test-driven development (разработка через тестирование) – методология разработки, подразумевающая написание тестовых сценариев до написания реализации.

Написание теста и его запуск (не выдает ложноположительный результат) – реализация метода – запуск теста, включая остальные тестовые сценарии – рефакторинг – финальный запуск теста и остальных тестируемых сценариев.

Шаблон для структурирования тестов – AAA (Arrange, act, assert) – разделяет написание тестовых методов на 3 логических шага:

*Arrange* (подготовка) – настройка всех необходимых объектов, данных и зависимостей, необходимых для выполнения теста. (Обозначается комментарием //given)

*Act* – выполняется тестируемый метод. (Обозначается комментарием //when)

*Assert* – проверка результата выполнения теста. (Обозначается комментарием //then)

**Mockito** – фреймворк для тестирования, который специализируется на создании и использовании mock-объектов. Он предоставляет инструменты для создания, настройки и использования моков, изолирует тестируемый код от зависимостей, создавая их имитации с контролируемым поведением.

**Test doubles** – термин, описывающий все виды объектов, используемых в тестировании для замены реальных зависимостей.

*Dummy* – этот объект, который используется для заполнения сигнатуры метода, но его содержание не важно в контексте тестирования (не влияют на поведение и результаты тестирования). В Mockito используются методы типа any().

*Fake* – объект, который реализует функциональность, но не является полноценной реализацией реальной зависимости, то есть повторяет функционал объекта, но в упрощенном виде. Он используется в тестировании для замены реальных зависимостей, которые не могут быть использованы при тестировании (медленная скорость, еще недоступен, не может использоваться из-за побочных эффектов).

*Stub* –объект, возвращающий заранее подготовленное значение в ответ на вызов метода (не имеет реальной логики). Два способа создания stub (см. ниже).

*Mock* – proxy-объект на основе передаваемого класса, имитирует реальный объект (методы с незафиксированным значением возвращают значение по умолчанию соответствующее типу данных), в отличие от stub имеет возможность верификации. Mockito.mock().

*Spy* – proxy-объект на основе передаваемого класса или объекта (хранит в себе ссылку объекта, на основе которого был создан), методы, для которых не был определен stub вызывают реализацию методов переданного объекта. Так же обладает возможностью верификации. Mockito.spy().

**Stubbing для методов:**

|  |  |
| --- | --- |
| Mockito.doReturn().when().method() | Mockito.when().then() |
| when – стандартный способ мокирования, используется при работе с методами, которые возвращают значение и не вызывают побочных эффектов (не выбрасывают исключений).  Do методы – используются в специфичных случаях, когда использование when невозможно:   1. Мокирование void методов 2. Нежелательны побочные эффекты при вызове метода (изменение состояния или выбрасывание исключения) – при работе со spy, т.к. when вызывает реализацию метода, а затем подменяет возвращаемое значение, do методы   \*when позволяет возвращать разные результаты на вызовы этого метода с помощью нескольких thenReturn(). | |

Интерфейс Answer – используется для того, чтобы задать более сложную логику для определения возвращаемого результата для stub, то есть не просто вернуть фиксированное значение а выполнить некую логику (thenAnswer() или doAnswer() в зависимости от способа stubbing), требует реализации метода <T> T answer(InvocationOnMock invok), входящий аргумент предоставляет возможности получения метаданных вызываемого метода.

**Верификация:** проверка кол-ва вызовов методов, порядка их вызова и того с какими параметрами вызывался метод.

*Verify()* – проверка того, что указанный метод на mock или spy объекте вызывался заданное кол-во раз, это кол-во задается с помощью VerificationMode (методы Mockito типа times(), atLeast(), never(), only() и др).

Mockito.*verify*(calculatorClass, Mockito.*times*(2)).getRandomIntOne();

//передаваемый мок-объект вызывает метод ровно getRandomIntOne() 2 раза

Параметры для верификации метода также либо указываются жестко, либо используется ArgumentMatchers.

*InOrder* – проверка порядка вызова методов. Для проверки создается объект класса InOrder с указанием объектов, затем на InOrder вызываются методы в ожидаемом порядке.

Mockito.*when*(calculatorClass.getRandomIntOne()).thenReturn(80);  
Mockito.*when*(calculatorClass.getRandomIntTwo()).thenReturn(80);

*assertTrue*(160 == calculator.sum());

//исполнили getRanOne, затем getRanTwo

InOrder inOrder = Mockito.*inOrder*(calculatorClass);  
inOrder.verify(calculatorClass).getRandomIntOne();  
inOrder.verify(calculatorClass).getRandomIntTwo();

// вызвали методы на объекте inorder в ожидаемом порядке

ArgumentCaptor – позволяет захватывать аргументы, передаваемые в вызываемые методы mock или spy. Создается объект ArgumentCaptor<> типа захватываемого значения с помощью аннотации @Captor или ArgumentCaptor.forClass(). Захватываются нужные аргументы в методе verify и проверяются на соответствие.

Mockito.*verify*(mock, Mockito.*times*(1)).sum(captor.capture(), captor.capture());  
*assertTrue*(100 == captor.getAllValues().get(0) + captor.getAllValues().get(1));

**Библиотека Mockito-junit-jupiter**

Библиотека для интеграции фреймворка Mockito с Junit – предоставляет возможности использования аннотаций для создания proxy-объектов.

Для внедрения расширения в Mocktio используется Extension model и аннотация @ExtendWith(MocktioExtension.class) или с помощью MockitoAnnotations.openMocks(this), что позволяет использовать аннотации библиотеки Mockito.

@Mock, @Spy, @Captor для создания соответствующих сущностей.

@InjectMocks – инициализирует объект и внедряет созданные mocks в тестируемый класс (mock-объект - это зависимость основного класса)

[Материалы Mockito](https://habr.com/ru/articles/444982/)