Курс «Создание модульных тестов с помощью JUnit»

Contents

1. Введение в JUnit

2. Основные компоненты JUnit

3. Группировка тестов,

4. Утверждения и предположения

5. Hamcrest matcher

6. Правила

7. дополнительные настройки тестового метода

8. Параметризация тестов

9. Runners

1 Intro to Junit

Краткое описание: Во введении кратко рассказывается о необходимости автоматизированного тестирования и использовании для этого библиотеки JUnit.

Одной из важнейших стадий в программировании является тестирование. При этом ручное тестирование отнимает у разработчика много времени: необходимость выполнять все наборы тестов (test cases) даже для малейшего изменения в коде крайне отрицательно сказывается на эффективности разработчика.

Поэтому большая часть ручных наборов тестов при проверке работоспособности приложения после изменений в коде чаще всего пропускается, что часто приводит к ошибкам в работе программы.

Чтобы уменьшить количество ошибок и быть уверенным что изменение кода не скажется на работоспособности программы (повлечет регрессию), используется подход, названный автоматизированным тестированием.

При этом подходе набор тестов представляет собой не описание сценария тестирования (который должен быть выполнен человеком), а набор запрограммированных действий, выполняющих этот сценарий тестирования.

Как правило, все автоматизированные тесты легко запустить на выполнение (из IDE или при сборке проекта) и по завершению работы формируется отчет, в котором видно, как повлияло изменение кода, были ли регрессии (отказавший/упавший тест)

Одним из типов автоматизированного тестирования является модульное (unit) тестирование.

От других типов тестирования (интеграционных и функциональных тестов) модульное тестирование отличается тем, что тестируются элементарные части программы: для ООП это будет отдельный класс или только отдельный метод конкретного класса.

JUnit – фреймворк с открытым исходным кодом для модульного (unit) тестирования кода написанного для платформы Java. Сейчас он является стандартом де-факто для модульного тестирования в Java.

Последняя версия JUnit – 4.11 и именно о возможностях этой версии JUnit будет в дальнейшем рассказываться.

Простой пример использования JUnit:

Предположим есть класс, который считает сумму двух чисел:

**class** Calculator {

**public** **int** sum(**int** a, **int** b) {

**return** a + b;

}

}

Чтобы проверить, что он считает правильно, напишем для него автоматизированный тест с помощью JUnit:

**import** org.junit.Assert;

**import** org.junit.Test;

**public** **class** TestClass {

**private** Calculator calc = **new** Calculator();

@Test

**public** **void** testMethod() {

Assert.*assertEquals*(calc.sum(1, 2), 3);

}

}

Если запустить этот тест на выполнение, тестовый метод testMethod будет выполнен успешно.

Если же поменять реализацию Calculator.sum, чтобы выполнялось не сложение а вычитание, то запуск теста на выполнение приведет к тому, что тестовый метод testMethod будет считаться отказавшим (failed).

Как видно из примера, JUnit старается быть легким в использовании фреймворком:

1) JUnit не требует чтобы тест-класс (TestClass в данном случае) наследовался от какого-либо Junit-специфичного класса.

2) На тест-метод (testMethod) также не налагается особых требований — достаточно добавить аннотацию @org.junit.Test. Каждый метод, аннотированный этой аннотацией, будет вызван в процессе тестирования

3) Assert.assertEquals сравнивает два числа (или объекта по методу equals – assertEquals переопределен чтобы поддерживать объекты и примитивные типы). Если два числа не равны, то бросается исключение java.lang.AssertionError и тест завершается неудачно

2. Основные компоненты тестового класса

Краткое описание: Вводится понятие тестового класса. Описывается структура тестового класса, основные его компоненты и правила формирования. Описываются жизненный цикл тестового класса и методы жизненного цикла.

Тестовый класс — это простой Java класс, который однако должен удовлетворять простым и ненавязчивым условиям:

1) Класс должен иметь конструктор без аргументов

2) Класс должен быть public

3) Также считается общепринятым соглашением оканчивать имя тестового класса на слово Test (например CalculatorTest)

- например, утилиты автоматической сборки часто полагаются на это соглашение при поиске тестовых классов

Когда программист создает тестовый класс с использованием JUnit он может включить в этот класс следующее:

1) Тестовые методы

2) Методы жизненного цикла теста

3) Конструктор

4) Просто поля и методы класса (т. к. тест-класс — это простой Java класс) - также как блоки статической и динамической инициализации

Тестовые методы

Основную роль при тестировании играет конечно же тестовые методы.

Каждый тестовый метод представляет собой реализацию тестового сценария (тестового случая). В примере с калькулятором сложение двух чисел еще не является тестовым сценарием — это конкретная функциональность программы, для которой может быть написано много тестовых сценариев. Например:

1) Сложение двух положительных чисел

2) Сложение положительного и отрицательного чисел

3) Сложение положительного числа и нуля

При этом, безусловно, все три случа можно описать в одном тестовом методе, но это плохая практика, т. к. не позволяет, взглянув на отчет о прошедших(succeeded)/отказавших(failed) тестах, сказать какие именно тестовые сценарии не были успешно выполнены.

Пример:

**public** **class** CalculatorAllInOneTest {

**private** Calculator calc = **new** Calculator();

@Test

**public** **void** testSum() {

Assert.*assertEquals*(calc.sum(1, 2), 3);

Assert.*assertEquals*(calc.sum(2, -1), 1);

Assert.*assertEquals*(calc.sum(2, 0), 2);

}

}

Такой тестовый метод также сложно поддерживать и читать.

Лучше разделить тестовые сценарии, выделив на каждый отдельный тестовый метод, как в следующем примере:

**public** **class** CalculatorManyTestMethodsTest {

**private** Calculator calc = **new** Calculator();

@Test

**public** **void** testSumPositives() {

Assert.*assertEquals*(calc.sum(1, 2), 3);

}

@Test

**public** **void** testSumPositiveAndNegative() {

Assert.*assertEquals*(calc.sum(-1, 2), 1);

}

@Test

**public** **void** testSumPositiveAndZero() {

Assert.*assertEquals*(calc.sum(2, 0), 2);

}

}

На данный момент такой подход может показаться излишним, так как проверка корректной работы занимает всего одну строку, но на практике маловероятно что тестовые сценарии будут такими же простыми.

И этот подход является одним из правил написания модульных тестов:

**Каждый тестовый метод должен реализовывать только один тестовый сценарий**

Простой метод класса становится тестовым методом, когда он помечается аннотацией @org.junit.Test.

Однако одного добавления аннотации недостаточно, необходимо также чтобы сигнатура метода соответствовала определенным условиям:

1) Метод должен быть public

2) Метод не должен быть static

3) Метод не должен иметь возвращаемое значение (всегда должен быть void)

4) Метод не должен иметь параметров (хотя есть исключения из этого правила)

С учетом всех этих ограничений, тестовый метод выглядит так:

@Test

**public** **void** nameOfMethod() {}

где меняется только имя метода

При этом:

а) Имя метода может быть любым

- Более ранняя версия JUnit (3.X) требовала чтобы все тестовые методы начинались с префикса test. Это было вызвано необходимостью фреймворку разграничивать тестовые и простые методы. JUnit 3.X старше чем Java 5 (и соответственно аннотации), поэтому наиболее простым решением было соглашение о именовании тестовых методов. В JUnit 4 нет необходимости начинать имя метода с префикса test, потому что тестовый ли метод определяется по наличию аннотации @Test

- но все же добавление префикса test рекомендуется, так как он выполняет роль глагола «тестировать». Например, testSumPositives читается как «протестировать сумму положительных»

- Хорошим тоном считается такое имя метода, которое объясняет тестовый сценарий, реализуемый в методе. Уже упоминавшийся testSumPositives представляет собой хорошее имя, а вот имен вида testUser, testAddUser1, testAddUser2 лучше избегать

б) Метод может быть абстрактным

При этом, конечно, сам тестовый класс также становится абстрактным и тестовые методы в нем без использования наследников класса запустить нельзя.

Но следующий пример вполне корректен:

**public** **abstract** **class** AbstractTest {

**protected** Calculator calc;

@Test

**public** **abstract** **void** testSumPositives();

}

**public** **class** SimpleCalculatorTest **extends** AbstractTest {

**public** SimpleCalculatorTest() {

calc = **new** Calculator();

}

@Override

**public** **void** testSumPositives() {

Assert.*assertEquals*(calc.sum(1, 2), 3);

}

}

Такой подход может быть удобен для тестирования иерархии классов построенных с помощью шаблонов Template Method или Strategy.

Чтобы класс считался тестовым, в нем должен быть хотя бы один тестовый метод.

В одном тестовом классе может быть много тестовых методов. При этом порядок их выполнения неизвестен (и не гарантируется, что он не изменится между прогонами на разных реализациях JVM). Отсюда вытекает следующее правило создания модульных тестов:

**Тестовый метод не должен зависеть от результата выполнения другого тестового метода**

При запуске тестового метода через JUnit результат выполнения тестового метода может быть одним из следующих:

1) Успешно выполнен (succeeded). Тест был успешно пройден. При разработке модульных тестов именно этот результат — желаемый результат выполнения всех тестов.

2) Пропущен. Иногда (особенно при рефакторинге) тесты могут становиться неактуальными и их можно отключить на время поочередного приведения их в порядок. Для этого достаточно аннотировать тестовый метод с помощью аннотации @org.junit.Ignore.

3) Отказавший (failed) – тест, который отказал из-за того, что при выполнении было брошено исключение java.lang.AssertionError. Это исключение обычно происходит если утверждения (assertions) становится ложным. Пример — Assert.assertEquals(1 + 1, 3).

В отчетах о выполненных тестах такие тесты помечаются как failures

4) Еще один тип отказавших тестов — тест, который отказал из-за брошенного исключения (кроме java.lang.AssertionError)

В отчетах о выполненных тестах такие тесты помечаются как errors

Следующий пример демонстрирует все 4 возможных результата выполнения:

**public** **class** QuestionTest {

@Test

**public** **void** testSumSucceeded() {

Assert.*assertEquals*(1+1, 2);

}

@Test

@Ignore

**public** **void** testSumIgnored() {

}

@Test

**public** **void** testSumFailedWithFailure() {

Assert.*assertEquals*(1, 2);

}

@Test

**public** **void** testSumFailedWithError() {

Assert.*assertEquals*(1, ((Integer) **null**).intValue());

}

}

Методы жизненного цикла теста

@Before/@After

Часто бывает выполнить некоторое действие до вызова тестового метода или после вызова.

JUnit предоставляет простой способ сделать это — пару аннотаций @Before/@After.

Аннотация @Before на методе означает, что этот метод необходимо выполнить до выполнения тестового метода, а аннотация @After – что этот метод необходимо выполнить после выполнения тестового метода.

Пример:

**public** **class** TestLifecycleMethods {

@Before

**public** **void** before() {

System.*out*.println("Перед выполнением тестового метода");

}

@After

**public** **void** after() {

System.*out*.println("После выполнения тестового метода");

}

@Test

**public** **void** testMethod() {

System.*out*.println("Тестовый метод выполняется");

}

}

При запуске тестовых методов из этого файла будет выведено следующее:

Перед выполнением тестового метода

Тестовый метод выполняется

После выполнения тестового метода

В классе может быть несколько методов аннотированных @Before (@After), но определенный порядок их выполнения не гаратируется. Поэтому код, размещенный в одном @Before методе, не должен зависеть от результата выполнения другого @Before метода.

Также не имеет значения, где в исходном коде находятся @Before/@After-методы — перед тестовым метод или после.

Как и с тестовыми методами, имя @Before / @After метода может быть любым и на сигнатуру метода налагаются те же требования, что и на тестовые методы (только public, не static, void, без параметров).

@BeforeClass/@AfterClass

Кроме выполнения действий до и после вызова тестового метода, JUnit 4 дает возможность написать методы, которые будут выполняться до создания экземпляра тестового класса и вызова любого тестового метода. Для этого предназначена аннотация @BeforeClass.

Если же необходимо вызвать метод после выполнения всех тестовых методов, то для этого служит аннотация @AfterClass

На @BeforeClass / @AfterClass методы распространяются все те же требования, что и на @Before / @After методы, кроме одного:

**@BeforeClass / @AfterClass методы обязаны быть статическими**

Пример:

**public** **class** TestLifecycleMethods2 {

@BeforeClass

**public** **static** **void** beforeClass() {

System.*out*.println("Перед выполнением всех тестовых методов");

}

@Before

**public** **void** before() {

System.*out*.println("Перед выполнением тестового метода");

}

@After

**public** **void** after() {

System.*out*.println("После выполнения тестового метода");

}

@AfterClass

**public** **static** **void** afterClass() {

System.*out*.println("После выполнения всех тестовых методов");

}

@Test

**public** **void** testMethod() {

System.*out*.println("Тестовый метод выполняется");

}

}

выведет:

Перед выполнением всех тестовых методов

Перед выполнением тестового метода

Тестовый метод выполняется

После выполнения тестового метода

После выполнения всех тестовых методов

Использование After/Before и AfterClass/BeforeClass

При выборе какую аннотацию использовать, стоит учитывать что методы аннотированные Before/After выполняются для каждого тестового метода, а AfterClass/BeforeClass – только один раз для всех тестовых методов в тестовом классе.

Поэтому AfterClass/BeforeClass удобно использовать для инициализации состояния тестового класса, которое будет неизменным для всех тестовых методов внутри данного тестового класса. Это также позволит тестам выполняться быстрее.

Before/After же обычно используется для подготовки контекста выполнения тестового метода если этот контекст может быть «загрязнен» выполнением другого тестового метода. Загрязненный контекст может привести к тому что тестовый метод, который должен выполниться успешно, будет работать на некорретных данных и упадет.

Пример разграничения: При тестировании работы с базой данных соединение можно создать в @BeforeClass и закрывать в @AfterClass, поскольку создание соединения — достаточно тяжелая по времени и ресурсам операции. Кроме того, выполнение тестовых методов не изменяет состояние соединения.

При этом выполнение SQL запросов не должно изменять состояние БД между запусками тестовых методов, поэтому в методе @Before нужно создавать транзакцию, а в методе @After – откатывать ее.

Конструктор

Тестовый класс должен иметь конструктор без аргументов. Если разработчик не создал конструктор для класса сам, компилятор Java создаст такой конструктор за него.

Если же разработчик создал конструктор (для инициализации полей например), то **конструктор должен быть без аргументов**.

Этот конструктор будет вызываться для создания экземпляра тестового метода, при этом:

**JUnit создает новый экземпляр тестового класса для выполнения каждого тестового метода**

Используя такой подход, JUnit гарантирует что контекст выполнения тестового метода не будет «загрязнен» результатами выполнения других методов.

Таким образом, инициализация экземпляра класса может быть выполнена:

1) поля могут инициализироваться при их определении (как в примерах с калькулятором — private Calculator calc = new Calculator())

2) Инициализация в конструкторе

3) Инициализация в блоке динамической инициализации

4) Инициализация в @Before методе

Эти способы инициализации описаны ниже в примере:

**public** **class** LifecycledTest {

**private** **static** String setState(String state) {

System.*out*.println(state);

**return** state;

}

**private** String state = *setState*("Инициализация при определении");

**public** LifecycledTest() {

state = *setState*("Инициализация в конструкторе");

}

{

state = *setState*("Инициализация в блоке динамической инициализации");

}

@Before

**public** **void** before() {

state = *setState*("Инициализация в @Before");

}

@Test

**public** **void** testMethod() {

}

}

Запуск тестового метода выведет следующее:

Инициализация при определении

Инициализация в блоке динамической инициализации

Инициализация в конструкторе

Инициализация в @Before

Предпочтительным при этом является использование @Before аннотации.

Поля и методы класса

Как Java класс, тестовый класс может содержать поля, методы, блоки статической и динамической инициализации.

Жизненный цикл тестового класса

Жизненный цикл прогона тестовых методов тестового класса тесно интегрирован со стандартным циклом загрузки класса и создания его экземпляров. Но в этой части будет описана лишь специфика методов жизненного цикла JUnit (Before/After, BeforeClass/AfterClass):

1) JUnit создает новый экземпляр тестового класса для прогонки каждого тестового метода

Но при этом сначала выполняются все @BeforeClass методы

2) JUnit поддерживает наследование. Правила здесь простые:

- сначала выполняются методы Before\* в родительском классе, а потом в наследнике

- After\* методы работают по симметричной схеме: сначала выполняются методы в наследнике, а потом в родительском классе

Понять как это происходит поможет следующий пример:

**public** **class** ParentLifecycledTest {

@BeforeClass

**public** **static** **void** beforeClassParent() {

System.*out*.println("Родитель: Перед выполнением всех тестовых методов");

}

@Before

**public** **void** beforeParent() {

System.*out*.println("Родитель: Перед выполнением тестового метода");

}

@After

**public** **void** afterParent() {

System.*out*.println("Родитель: После выполнения тестового метода");

}

@AfterClass

**public** **static** **void** afterClassParent() {

System.*out*.println("Родитель: После выполнения всех тестовых методов");

}

}

**public** **class** DescendantLifecycledTest **extends** ParentLifecycledTest {

@BeforeClass

**public** **static** **void** beforeClass() {

System.*out*.println("Наследник: Перед выполнением всех тестовых методов");

}

@Before

**public** **void** before() {

System.*out*.println("Наследник: Перед выполнением тестового метода");

}

@After

**public** **void** after() {

System.*out*.println("Наследник: После выполнения тестового метода");

}

@AfterClass

**public** **static** **void** afterClass() {

System.*out*.println("Наследник: После выполнения всех тестовых методов");

}

@Test

**public** **void** testMethodDescendant() {

System.*out*.println("Наследник: Тестовый метод выполняется");

}

}

При запуске тестового метода в DescendantLifecycledTest, будет выведено следующее:

Родитель: Перед выполнением всех тестовых методов

Наследник: Перед выполнением всех тестовых методов

Родитель: Перед выполнением тестового метода

Наследник: Перед выполнением тестового метода

Наследник: Тестовый метод выполняется

Наследник: После выполнения тестового метода

Родитель: После выполнения тестового метода

Наследник: После выполнения всех тестовых методов

Родитель: После выполнения всех тестовых методов

Группировка тестов

При внесении изменений в уже написанную функциональность, всегда существует вероятность внесения багов. Автоматические тесты (в том числе и модульные) помогают проверить, не было ли регрессий. При этом запуск всех автоматизированных тестов может быть достаточно долгим. Но поскольку часто набор всех тестов можно разделить на слабосвязанные группы, для проверки работоспособности изменяемой функциональности (модуля приложения) достаточно запустить тесты из группы, тестирующей этот модуль.

Например, если в приложении можно выделить модуль безопасности (работа с пользователями, группами и т. д.), модуль создания заказов и модуль оплаты, то для проверки корректности создания пользователя достаточно запустить только те тесты которые относятся к модулю безопасности.

JUnit предоставляет следующие возможности для группировки тестов:

1) Наборы тестов (Test Suites)

2) Категории (Categories)

Для понимания того, как JUnit определяет что есть тестовый класс и что есть категория и набор тестов, необходимо познакомиться с еще одним типом компонентов, используемых в JUnit – runner (примерный перевод — пусковик, запускальщик, ввиду неблаговидности названия будем использовать слово runner).

Runner собственно и запускает тестовый класс. У JUnit есть несколько стандартных runner'ов, среди них org.junit.runners.JUnit4 – стандартный (по умолчанию) runner. Чтобы сказать что тест запускается с помощью определенного runner'а, необходимо проаннотировать класс аннотацией @RunWith и передать в качестве параметра runner класс.

Таким образом, объявление класса как

public class SimpleTest {}

аналогично следующему

@RunWith(JUnit4.class)

public class SimpleTest {}

Для запуска же наборов тестов и категорий есть свои runner'ы

Наборы тестов (Test Suites)

Наборы тестов позволяют группировать тесты по какому-либо признаку, создавая иерархии тестов:

Набор Тестов 1 (Test Suite)

→ Набор Тестов 2 (Test Suite)

→ Тест-класс 1 (Test Class)

→ Тест-класс 2 (Test Class)

Представим, что есть класс для тестирования и набор тестов для него:

**class** Calculator {

**public** **int** divide(**int** a, **int** b) {

**if** (b == 0) {

**throw** **new** IllegalArgumentException("Parameter divider should not be 0");

}

**return** Math.*round*(a / b);

}

**public** **long** sqrt(**int** a) {

**if** (a < 0) {

**throw** **new** IllegalArgumentException("Parameter should be positive or 0");

}

**return** Math.*round*(Math.*sqrt*(a));

}

}

**public** **class** CalculatorDivideTest {

**private** Calculator calc = **new** Calculator();

@Test

**public** **void** testDividePositives() {

Assert.*assertEquals*(calc.divide(10, 2), 5);

}

@Test

**public** **void** testDividePositiveAndNegative() {

Assert.*assertEquals*(calc.divide(-3, 3), -1);

}

@Test(expected = IllegalArgumentException.**class**)

**public** **void** testDivideByZero() {

calc.divide(2, 0);

}

}

**public** **class** CalculatorSqrtTest {

**private** Calculator calc = **new** Calculator();

@Test

**public** **void** testSqrtPositive() {

Assert.*assertEquals*(calc.sqrt(9), 3);

}

@Test

**public** **void** testSqrtZero() {

Assert.*assertEquals*(calc.sqrt(0), 0);

}

@Test(expected = IllegalArgumentException.**class**)

**public** **void** testSqrtNegative() {

calc.sqrt(-2);

}

}

Здесь была введена новая конструкция - @Test(expected = SomeException.class). При выполнении такого тестового метода JUnit будет ожидать, что при выполнении метода будет брошено исключении объявленного типа. Только в этом случае тестовый метод будет считаться успешно выполненным (succeeded). Если же при выполнении тестового метода исключение брошено не было (или было, но другого типа), то тест считается отказавшим (failed).

Чтобы организовать эти тестовые классы в набор тестов, необходимо:

1) Создать класс для хранения списка тестовых классов. Класс необходим только для того аннотировать его необходимыми для набора тестов аннотациями

2) Аннотировать созданный класс с помощью аннотации

@RunWith(**org.junit.runners.Suite.class**)

В качестве runner здесь используется Suite.class – runner, который знает как запускать набор из тестовых классов

3) Аннотировать созданный класс с помощью аннотации @Suite.SuiteClasses и перечислить в параметре аннотации список тестовых классов которые будут входить в этот набор тестов

К примеру, все тесты из примера выше можно запустить в одном наборе, создав следующий класс:

@RunWith(Suite.**class**)

@Suite.SuiteClasses({

CalculatorDivideTest.**class**,

CalculatorSqrtTest.**class**

})

**public** **class** AllOperationsSuite {

}

При этом, если указывать в списке тестовых классов в качестве тестового класса класс набора тестов, то можно организовывать тесты в иерархии произвольной вложенности. Следующий пример аналогичен классу AllOperationsSuite выше:

@RunWith(Suite.**class**)

@Suite.SuiteClasses({

CalculatorDivideTest.**class**

})

**public** **class** BasicOperationsSuite {

}

@RunWith(Suite.**class**)

@Suite.SuiteClasses({

CalculatorSqrtTest.**class**,

BasicOperationsSuite.**class**

})

**public** **class** AllOperations2Suite {

}

Таким образом можно распределять классы в мелкие наборы тестов (по функциональности), а уже мелкие наборы объединять в более крупные (по субмодулям и подсистемам). Например, можно создать набор тестов (test suite) для работы с пользователями, набор тестов для работы с ролями, набор тестов для проверки авторизации и объединить все три набора в один большой — набор тестов для модуля безопасности.

Класс набора тестов (такой как AllOperationsSuite) имеет следующие особенности:

1) Это должен быть именно класс, не интерфейс

2) В нем можно определить методы @Before/@After аналогично тестовым классам:

@RunWith(Suite.**class**)

@Suite.SuiteClasses({

CalculatorMultiplyTest.**class**,

SumSubSuites.**class**

})

**public** **class** AllOperations2Suite {

@BeforeClass

**public** **static** **void** beforeAllTests() {

System.*out*.println("Перед всеми тестами");

}

@AfterClass

**public** **static** **void** afterAllTests() {

System.*out*.println("После всех тестов");

}

}

Метод beforeAllTests будет выполняеться перед всеми тестами, включенными в набор, а метод afterAllTests – после всех. Эту возможность удобно использовать если необходимо создать или загрузить некоторые данные от которых зависят все тесты в наборе и которые не изменяются в тестах. Например в модуле безопасности это может быть создание тестового пользователя, на наличие которого в контексте безопасности рассчитывают все тесты.

Конструкторы, блоки динамической инициализации, @Before / @After методы и любые другие (даже тестовые) методы игнорируются и не выполняются JUnit (поскольку он вообще не создает экземпляров этого класса) .

4) С помощью набора тестов (test suite) объединять в группы можно только тестовые классы, но не тестовые методы.

Категории

Хотя наборы тестов (test suites) позволяют организовывать тесты в группы, гибкость настройки включения/исключения ограничивается только уровнем тестового класса.

Категории (Categories) – функциональность JUnit, которая расширяет и дополняет возможности наборов тестов (test suites). Категории позволяют помечать тестовые классы и тестовые методы, относя их к определенным группам.

На текущий момент (JUnit 4.11) категории пока считаются экспериментальной функциональностью и могут быть не до конца отлаженными.

Чтобы создать и запустить на выполнение только определенную группу тестов, необходимо:

1) Создать новую категорию. Категория, представляет собой по сути имя для группы тестов. Но в отличии от других тестовых фреймворков (TestNG например), имя группы — это не строка, а имя класса (или интерфейса). Например:

**public** **interface** BasicCases {

}

Или

**public** **class** BasicCases {

}

Этот класс служит просто маркером категории, поэтому достаточно использовать интерфейс. Использование класса (интерфейса) в качестве маркера вместо строкового маркера (простой строки — имени категории) имеет как достоинства так и недостатки. К недостаткам можно отнести избыточность такого подхода (целый класс используется для создания всего одной категории — увеличивается время компиляции), но с другой стороны, использование классов позволяет избежать опечатки в имени категории, поскольку имена проверяются при компиляции.

Никаких дополнительных аннотаций или наследования от определенных классов не требуется

2) Аннотировать тестовые классы или тестовые методы с помощью аннотации @org.junit.experimental.categories.Category, перечислив категории, к которым относится этот класс (для этого понадобятся классы, созданные на шаге 1). При этом аннотация Category объявлена как:

**public** **@interface** Category {

Class<?>[] value();

}

Поэтому передается именно список категорий и один тестовый методы (или тестовый класс) может быть в нескольких группах тестовы.

Пример:

@Category({BasicCases.**class**})

**public** **class** Test {

@Test

**public** **void** test1() {

}

@Category({ExceptionalCases.**class**})

@Test()

**public** **void** test2() {

}

}

При этом если категория применяется на уровне тестового класса, то все тестовые методы включаются в эту категорию. Именно поэтому в примере хоть на тестовом методе test1 нет аннотации @Category, он все равно включен в категорию Basics, а тестовый метод test2 включен в две категории — Basics и ExceptionalCases.

3) Создать класс для хранения списка тестовых классов. Его роль будет точно такой же как и для набора тестов (test suite) – только чтобы прописать соответствующие аннотации, поэтому также будем ссылаться на него как «класс набора тестов»

Как и в случае с набором тестов, этот класс не нужно наследовать от какого-либо специфичного для JUnit родительского класса. И так же как и с набором тестов, класс может содержать методы аннотированные с помощью @Before / @After

Пример:

**public** **class** BasicTests {

}

4) Аннотировать созданный класс с помощью аннотации

@RunWith(**org.junit.experimental.categories.Categories.class**)

В качестве runner здесь используется **Categories**.class – этот runner является наследником runner'а Suite.class, поэтому он также распознает аннотацию @SuiteClasses и умеет запускать набор тестов. Название пакета намекает на то что функциональность категорий в JUnit экспериментальна и может быть не до конца отлажена. Но все же рекомендуется использовать категории как весьма полезную и довольно стабильную возможность JUnit.

Пример:

@RunWith(Categories.**class**)

**public** **class** BasicTests {

}

5) Аннотировать созданный класс с помощью аннотации @Suite.SuiteClasses и перечислить в параметре аннотации список тестовых классов которые будут входить в этот набор тестов.

Так как категории расширяют функционал набора тестов, все правила применяемые для аннотации @SuiteClasses справедливы и здесь.

Более того, если остановиться на этом шаге, то созданный класс будет работать ровно как класс набора тестов (test suite).

Пример:

@RunWith(Categories.**class**)

@SuiteClasses( {

CalculatorDivideTest.**class**,

CalculatorSqrtTest.**class**

})

**public** **class** BasicTests {

}

6) Аннотировать созданный класс с помощью аннотаций @org.junit.experimental.categories

.Categories.IncludeCategory или @org.junit.experimental.categories

.Categories.ExcludeCategory

Пример:

@RunWith(Categories.**class**)

@IncludeCategory(BasicCases.**class**)

@SuiteClasses( {

CalculatorDivideTest.**class**,

CalculatorSqrtTest.**class**

})

**public** **class** BasicTests {

}

@IncludeCategory служит для включения в список запускаемых тестов только те тестовые методы, которые: а) объявлены в тестовых классах перечисленных в @SuiteClasses б) включены в категорию, указанную в @IncludeCategory

Предположим, что тестовые классы CalculatorDivideTest и CalculatorSqrtTest были изменены чтобы тестовые методы относились к категориям BasicTests и ExceptionalTests:

**public** **class** CalculatorDivideTest {

**private** Calculator calc = **new** Calculator();

@Category({BasicCases.**class**})

@Test

**public** **void** testDividePositives() {

Assert.*assertEquals*(calc.divide(10, 2), 5);

}

@Test

**public** **void** testDividePositiveAndNegative() {

Assert.*assertEquals*(calc.divide(-3, 3), -1);

}

@Category({ExceptionalCases.**class**})

@Test(expected = IllegalArgumentException.**class**)

**public** **void** testDivideByZero() {

calc.divide(2, 0);

}

}

@Category({BasicCases.**class**})

**public** **class** CalculatorSqrtTest {

**private** Calculator calc = **new** Calculator();

@Test

**public** **void** testSqrtPositive() {

Assert.*assertEquals*(calc.sqrt(9), 3);

}

@Test

**public** **void** testSqrtZero() {

Assert.*assertEquals*(calc.sqrt(0), 0);

}

@Category({ExceptionalCases.**class**})

@Test(expected = IllegalArgumentException.**class**)

**public** **void** testSqrtNegative() {

calc.sqrt(-2);

}

}

Таким образом, можно запустить только тесты, относящиеся к категории базовых тестов с помощью следующего класса набора тестов:

@RunWith(Categories.**class**)

@IncludeCategory(Basics.**class**)

@SuiteClasses( {

CalculatorDivideTest.**class**,

CalculatorSqrtTest.**class**

})

**public** **class** BasicTests {

}

Особенности использования @IncludeCategory:

1) Аннотация объявлена как:

**public** **@interface** IncludeCategory {

**public** Class<?> value();

}

Поэтому можно включить только одну категорию для конкретного набора тестов

2) Тестовый метод будет включен в список запускаемых тестов, если он входит хотя бы в одну категорию. Т.е. если тестовый метод аннотирован как @Category({BasicTests.class, ExceptionalTests.class}), достаточно написать @IncludeCategory(BasicTests.class), чтобы он был включен.

3) Если аннотация @IncludeCategory не указана на классе набора тестов, то запускаются все тестовые методы, вне зависимости от того, в какие категории они включены (использование @IncludeCategory пока не рассматривается)

То, что для определенного класса набора тестов можно указать только одну категорию (особенность #1 выше), налагает серьезные ограничения (особенно если категорий много и они не пересекаются). При этом аннотация @IncludeCategory может быть полезной для быстрого прогона только определенного набора тестовых методов. Предположим, есть следующий тест:

**interface** UnitTest {}

**interface** IntegrationTest {}

**public** **class** UserManagementTest {

@Category({UnitTest.**class**})

**public** **void** testCreateUser() {

}

@Category({IntegrationTest.**class**})

**public** **void** testCreateUserInExternalSystem() {

}

}

Здесь создается две категории — UnitTest для модульных тестов и IntegrationTest для интеграционных. При этом интеграционные тесты явно будут гораздо медленнее модульных тестов. И если при изменении функциональности создания пользователя оба теста отказали, то нет смысла запускать весь набор тестов (включая медленные интеграционные), пока модульные тесты не выполняются успешно. Поэтому с помощью аннотации @IncludeCategory можно временно подключить только модульные тесты и запускать только их.

Также тестовые методы и тестовые классы входящие в определенную категорию, можно исключать из списка запускаемых тестов с помощью аннотации @ExcludeCategory

@RunWith(Categories.**class**)

@IncludeCategory(ExceptionalTests.**class**)

@SuiteClasses( {

CalculatorDivideTest.**class**,

CalculatorSqrtTest.**class**

})

**public** **class** BasicTests {

}

Особенности использования аннотации:

1) Как и @IncludeCategory, @ExcludeCategory позволяет исключить только одну категорию

2) Может применяться совместно с @IncludeCategory

При этом указать в @IncludeCategory и @ExcludeCategory одну и ту же категорию нельзя — JUnit выдаст ошибку. Но если указать в @ExcludeCategory другую категорию, и тестовый метод относится к обеим категориям, то в список запускаемых тестов он включен не будет (@ExcludeCategory имеет приоритет)

Пример:

@RunWith(Categories.**class**)

@IncludeCategory(BasicCases.**class**)

@ExcludeCategory(ExceptionalCases.**class**)

@SuiteClasses( {

CalculatorDivideTest.**class**,

CalculatorSqrtTest.**class**

})

**public** **class** BasicTests {

}

# Утверждения и предположения

# Утверждения

Когда разработчик создает тест, он создает его с одной целью — проверить что тестируемая функциональность (класс) работает согласно заложенной логике. Если в будущем при работе с этим классом логика поменяется (например разработчик проводит рефакторинг или исправляет баг в логике) и в нее закрадется скрытая ошибка, то хорошо написанный тест должен отказать (fail), сигнализируя о проблеме.

При написании какой-либо функциональности разработчик сначала дробит ее на сценарии использования (например для системы безопасности создание нового пользователя — это один сценарий, а вход пользователя в систему — другой сценарий). Далее разработчик для каждого сценария использования придумывает тестовые сценарии (тестовые случаи). Для сценария использования «вход пользователя в систему» тестовыми сценариями могут быть «Пользователь с заблокированной учетной записью входит в систему», «Пользователь с несуществующим логином пытается войти в систему». Как правило, на каждый тестовый сценарий создается свой отдельный тестовый метод.

Описании тестового сценария (как и любое взаимодействие с создаваемой системой) можно представить в виде двухшагового алгоритма: 1) тест вызывает тестируемый модуль (метод тестируемого класса), передавая входные параметры 2) тестируемый модуль обрабатывает входную информацию в соответствии с заложенным алгоритмом и возвращает выходную информацию

Задача разработчика при создании теста:

1) Подготовить такие входные данные, чтобы модуль при вызове выполнял именно тот тестовый сценарий, который разработчик хочет протестировать

2) Проверить, что выходные данные соответствуют тем, которые разработчик ожидает получить согласно внутренней логике модуля.

При этом под выходными данными понимается:

1) Возвращаемое методом значение

2) Бросаемое методом исключение

3) Внутреннее состояние модуля

Для того, чтобы проверить корректность выходных данных, разработчик строит утвеждения. Например для тестового сценария логина заблокированного пользователя утверждения могут быть:

- Заблокированный пользователь не может успешно войти в систему

- Заблокированному пользователю выводится сообщение «Извините, ваша учетная запись заблокирована»

JUnit для реализации утверждний предлагает поспользоваться классом org.junit.Assert.

Этот класс содержит множество методов начинающихся со слова assert (утверждаю). Эти методы использует разработчик в тестовом методе, чтобы написать утверждение. При выполнении утверждения, если оно верно, то выполнение продолжится. А если ложно, то JUnit сгенерирует исключение java.lang.AssertionError, которое приведет к тому что тест откажет (failed).

Пример:

**public** **class** TestClass {

**private** Calculator calc = **new** Calculator();

@Test

**public** **void** testMethod() {

Assert.*assertEquals*(calc.sum(1, 2), 3);

}

}

**class** Calculator {

**public** **int** sum(**int** a, **int** b) {

**return** a + b;

}

}

Здесь при тестировании класса Calculator мы утверждаем, что при сложении двух положительных чисел не ожидается никаких исключений и числа будут успешно сложены по правилам операции сложения. Метод Assert.assertEquals(expected, actual) позволяет построить утверждение что два переданных в него параметра равны друг другу.

Эти методы могут использоваться по-разному:

1) Импортом класса Assert и вызовом статического метода утверждения.

Например:

**import** org.junit.Assert;

Assert.*assertEquals*(calc.sum(1, 2), 3);

2) Либо можно воспользоваться статическим импортом метода и использовать более краткую форму:

**import static** org.junit.Assert.assertEquals;

*assertEquals*(calc.sum(1, 2), 3);

Последняя форма более краткая и читабельная. При этом если используется много статических методов из класса Assert, имеет смысл импортировать все методы:

**import static** org.junit.Assert.\*;

*assertEquals*(calc.sum(1, 2), 3);

Также каждый из методов утверждения (а их много — assertEquals, assertTrue и т. д.) имеет две формы. На примере контракта assertEquals:

1) **static** **public** **void** assertEquals(**long** expected, **long** actual)

2) **static** **public** **void** assertEquals(String message, **long** expected, **long** actual)

Первый вариант метода не содержит сообщения заданного разработчиком и если утверждение станет ложным, тест откажет с сообщением, сгенерированным JUnit.

Например:

**public** **class** TestClass {

@Test

**public** **void** testMethod() {

Assert.*assertEquals*(1, 3);

}

}

Выполнение тестового метода testMethod приведет к отказу с сообщением вида: expected:<1> but was:<3>

Это сообщение подсказывает что первый аргумент (ожидаемое значение 1) не равен реальному (как правило вычисляемому значению (3).

Сгенерированное сообщение довольно детальное, но может не описывать причину отказа теста.

Пример:

**public** **class** TestClass {

**private** **static** **final** **int** *PRODUCT\_COUNT* = 3;

@Test

**public** **void** testMethod() {

Assert.*assertEquals*("Количество товаров отличается от ожидаемого.",

*PRODUCT\_COUNT, getShopProductCount()*);

}

}

Если тест откажет, сообщение будет «Количество товаров отличается от ожидаемого. expected:<3> but was:<2>» и быстрее поможет разобраться в причине отказа теста.

Все методы утверждения в классе Assert имеют форму:

**public** **static** **void** nameOfAsserttion(Parameters...)

поэтому в дальнейших примерах контракт метода будет обрезан до формы nameOfAsserttion(Parameters...)

Методы утверждения в классе Assert делятся на несколько групп:

1) Логические утверждения.

Это утверждения вида «Утверждаю, что вычисленное значение истинно» или «Утверждаю, что вычисленное значение ложно»

Это методы:

- assertTrue

assertTrue(String message, **boolean** condition)

assertTrue(**boolean** condition)

Это утверждение будет ложным если переданный логический (boolean) параметр содержит false значение

- assertFalse

assertFalse(String message, **boolean** condition)

assertFalse(**boolean** condition)

Это утверждение будет ложным если переданный логический (boolean) параметр содержит true значение

Пример:

**public** **class** LogicalAssertionTest {

@Test

**public** **void** testUser() {

Assert.*assertTrue*(userIsValid("testUser"));

Assert.*assertFalse*(userIsBlocked("testUser"));

}

}

Безусловно, эти утверждения можно использовать, чтобы выразить все остальные утверждения (пример assertTrue(3 == getUserOrderCount())), но при этом теряется контекст и сообщения в этом случае не подскажут, что ожидаемое значение 3, а реальное значение 2.

2) Утверждения сравнения.

Утверждения сравнения - «Утверждаю, что ожидаемое значение X равно вычисленному значению Y».

Это множественные перегруженные методы с именем assertEquals и двумя (тремя — если нужно передать сообщение) параметрами (ожидаемым и реальным).

Эти методы перегружены чтобы корректно сравнивать примитивные типы (простая операция сравнения подойдет) и объекты (вызывается метод equals).

Есть следующие методы сравнения:

|  |  |
| --- | --- |
| Тип параметра | Контракт метода |
| byte, short, int, long | **assertEquals**(**long** expected, **long** actual)  **assertEquals**(String message, **long** expected, **long** actual)  Меньшие типы (byte, short, int) будут автоматически приведены к типу long. Сравнение выполняется простым оператором сравнения (==) |
| float | **assertEquals**(**float** expected, **float** actual, **float** delta)  **assertEquals**(String message, **float** expected, **float** actual, **float** delta)  Смотри описание для double |
| double | **assertEquals**(**double** expected, **double** actual, **double** delta)  **assertEquals**(String message, **double** expected,  **double** actual, **double** delta)  Поскольку double и float – типы с плавающей точкой и при их представлении и работе с ними возможны ошибки округления, то JUnit позволяет сравнить эти числа с определенной точностью.  К примеру, чтобы проверить утверждение 1/3 = 0.(3), нам достаточно проверить что 1/3 приблизительно равно 0.33. Самый простой способ для этого — assertTrue(Math.abs(1/3 – 0.33) <= delta). Но JUnit уже использует эту логику внутри метода assertEquals(float/double, float/double, float/double), поэтому достаточно написать:  assertEquals(1.0/3, 0.33, 0,004)  Чтобы вычислить дельту (третий параметр), нужно отнять от реального результата 0.(3) округленный результат 0.33 и округлить полученное значение (0.00(3) == 0.004) |
| Любой объект | **assertEquals**(Object expected, Object actual)  **assertEquals**(String message, Object expected, Object actual)  Сравнение осуществляется путем вызова метод equals у параметра expected и передачи в качестве параметра объект actual. Для строк (java.lang.String) также будет вызван этот метод |

Если два объекта не равны, тест откажет с сообщением:

expected:<строковое представление expected> but was:<строковое представление actual>

где «строковое представление expected/actual» - это результат выполнения метода toString у expected / actual (если expected/actual null, то JUnit корректно обработает этот случай и выведет null). Два null объета считаются равными.

Также как есть утверждение вида «Утверждаю, что ожидаемое значение X равно вычисленному значению Y», есть и утверждение вида «Утверждаю, что ожидаемое значение X НЕ равно вычисленному значению Y».

Методы для неравенства двух значений схожи:

|  |  |
| --- | --- |
| Тип параметра | Контракт метода |
| byte, short, int, long | **assertNotEquals**(**long** expected, **long** actual)  **assertNotEquals**(String message, **long** expected, **long** actual)  Меньшие типы (byte, short, int) будут автоматически приведены к типу long. Сравнение выполняется простым оператором сравнения (!=) |
| float, double | **assertEquals**(**double** expected, **double** actual, **double** delta)  **assertEquals**(String message, **double** expected,  **double** actual, **double** delta)  Ожидаемый и реальный параметр должны быть отличаться (с учетом дельты). Может быть воспринята как инвертированная форма assertEquals ( Math.abs(1/3 – 0.33) > delta) |
| Любой объект | **assertEquals**(Object expected, Object actual)  **assertEquals**(String message, Object expected, Object actual)  Сравнение осуществляется путем вызова метод equals у параметра expected и передачи в качестве параметра объект actual. Для строк (java.lang.String) также будет вызван этот метод |

Если два объекта равны, тест откажет с сообщением:

Values should be different. Actual: <строковое представление actual>

Также есть утверждения для сравнения идентичности двух объектов. В отличии от assertEquals, эти методы сравнивают не значения (используя метод equals у ожидаемого объекта), а ссылки (используя оператор ==):

1) Утверждение идентичности двух объектов:

**assertSame**(String message, Object expected, Object actual)

**assertSame**(Object expected, Object actual)

Утверждение будет верно только если ожидаемый и реальный параметры указывают на одну область памяти. Например в при прогоне такого теста он откажет

Assert.*assertSame*(**new** Double(2.14), **new** Double(2.14))

При этом за счет интернирования строк следующее утверждение будет истинным:

Assert.*assertSame*("Hello", "Hello")

2) Утверждениe отличия двух объектов:

**assertNotSame**(String message, Object expected, Object actual)

**assertNotSame**(Object expected, Object actual)

Это инвертированная версия предыдущего утверждения, оно будет верно только если ожидаемый и реальный параметры НЕ указывают на одну область памяти

3) Утверждения сравнения для массивов.

Очень похожи на простые утверждения сравнения, но они поэлементно сравнивают массивы, выводя сообщение о том, какие элементы отсутствуют.

Есть следующие методы сравнения:

|  |  |
| --- | --- |
| Тип параметра | Контракт метода |
| byte | **assertArrayEquals**(**byte**[] expecteds, **byte**[] actuals)  **assertArrayEquals**(String message, **byte**[] expecteds,  **byte**[] actuals) |
| short | **assertArrayEquals**(**short**[] expecteds, **short**[] actuals)  **assertArrayEquals**(String message, **short**[] expecteds,  **short**[] actuals) |
| int | **assertArrayEquals**(**int**[] expecteds, **int**[] actuals)  **assertArrayEquals**(String message, **int**[] expecteds,  **int**[] actuals) |
| long | **assertArrayEquals**(**long**[] expecteds, **long**[] actuals)  **assertArrayEquals**(String message, **long**[] expecteds,  **long**[] actuals) |
| char | **assertArrayEquals**(**char**[] expecteds, **char**[] actuals)  **assertArrayEquals**(String message, **char**[] expecteds,  **char**[] actuals) |
| float | **assertArrayEquals**(**float**[] expecteds, **float**[] actuals, **float** delta)  **assertArrayEquals**(String message, **float**[] expecteds,  **float**[] actuals, **float** delta) |
| double | **assertArrayEquals**(**float**[] expecteds, **float**[] actuals, **float** delta)  **assertArrayEquals**(String message, **float**[] expecteds,  **float**[] actuals, **float** delta) |
| Любой объект | **assertArrayEquals**(String message, Object[] expecteds,  Object[] actuals)  **assertArrayEquals**(Object[] expecteds, Object[] actuals) |

Аналогично утверждениям сравнения объектов assertEquals, примитивные типы сравниваются при помощи оператора сравнения ==, а объекты — с помощью метода equals. При сравнении float и double точно так же используется дельта (погрешность).

Основное отличие от assertEquals в том, что если два массива (ожидаемый и реальный) различаются, JUnit укажет на различие:

1) Если массивы различаются длиной, то JUnit выдаст:

array lengths differed, expected.length=<expected> actual.length=<actual>

2) Если массивы одинаковой длины, то JUnit берет по одному элементу из каждого массива и сравнивает. Если равны, берется следующая пара элементов, если не равны, то тест отказывает с сообщением:

arrays first differed at element [<element index>]; expected:<expected[index]> but was:<actual[index]>

Например, следующее утверждение приведет к отказу теста:

Assert.*assertArrayEquals*(**new** **int**[] {2, 3, 5, 7}, **new** **int**[]{3, 1, 6, 7});

Сообщение при этом будет

arrays first differed at element [0]; expected:<2> but was:<3>

4) Утверждения присутствия значения

Если необходимо сравнить реальное (вычисленное) значение с null, то можно воспользоваться утверждением сравнения (например, assertEquals(null, calculateValue())).

Но поскольку необходимость сравнения с null возникает очень часто, в JUnit были добавлены специальные утверждения:

1) Утверждение проверки отсутствия значения:

**assertNull**(String message, Object object)

**assertNull**(Object object)

Если переданный параметр не null, то тест откажет с сообщением:

expected null, but was:<actual value>

2) Утверждение проверки присутствия значения:

**assertNotNull**(String message, Object object)

**assertNotNull**(Object object)

Если переданный параметр null, то тест откажет.

5) Отдельно стоит функционал, который врядли можно отнести к утверждениям. Это методы:

**fail**(String message)

**fail**()

Как только вызывается один из этих методов, тест отказывает (сразу же безусловно бросается исключение AssertionError).

Эти методы могут быть довольно полезны, например, для проверки генерации ислючений:

@Test

**public** **void** testFailMethod() {

**try** {

**int**[] array = **new** **int**[] {0, 1, 2};

System.*out*.println(getElementNumber100(array));

Assert.*fail*();

} **catch** (IndexOutOfBoundsException e) {

// nothing to do, test was executed correctly

}

}

**private** **int** getElementNumber100(**int**[] array) {

**return** array[100];

}

Assert.fail() в данном примере в случае корректной работы программы никогда не должен быть вызван. Если же выполнение дошло до вызова данного метода, значит метод getElementNumer100 реализован некорретно, и это ошибка (баг).

6) Компонуемое утверждение

Компонуемое утверждение отличается от других тем что разработчик сам настраивает логику проверки корректности утверждения.

Контракт этого метода выглядит так:

**public** **static** <T> **void** assertThat(String reason, T actual, org.hamcrest.Matcher<? **super** T> matcher)

**public** **static** <T> **void** assertThat(T actual, org.hamcrest.Matcher<? **super** T> matcher)

Этот тип утверждения наиболее гибок, поскольку позволяет не только реализовать все остальные утверждения, но и составить новые с помощью вызовов статических методов, формирующих DSL (Domain-Specific Language – язык, который предназначен для определенной области применения и хорошо подходящий для нее).

Вся эта гибкость реализуется благодаря матчеру (matcher – org.hamcrest.Matcher). Благодаря ему можно не только выразить Assert.*assertEquals*(9, var) как Assert.*assertThat*(var, *equalTo*(9)), но и построить утверждения, для которого нет стандартных средств в JUnit. Например, следующее утверждение проверяет, что в коллекции строк values есть строка «Two»

:

Assert.*assertThat*(values, *contains*("Two"));

Предположения

Иногда бывает необходимо выполнять тест только если выполняются некоторые условия (например, если тест зависит от окружения, в котором выполняется).

В таком случае удобно использовать предположения. Предположение в JUnit выглядит как вызов статических методов org.junit.Assume.assume\*. При этом если предположение истинно, то тест продолжит выполнение, иначе — будет игнорирован. В этом ключевое отличие от утверждения. Если утверждение ложно, тест отказывает. Если предположение ложно, тест прерывается и помечается как проигнорированный.

Пример использования предположения:

**public** **class** AssumptionsTest {

@Test

**public** **void** testAssume() {

Assume.*assumeTrue*(System.*getProperty*("path.separator").equals(":"));

}

}

Этот тестовый метод будет проигнорирован на Windows-системе, но завершен успешно на Unix-системах.

Есть несколько форм предположений:

|  |  |
| --- | --- |
| Контракт | Комментарий |
| assumeTrue(**boolean** b)  assumeTrue(String message, **boolean** b) | Тест будет проигнорирован если параметр == false |
| assumeFalse(**boolean** b)  assumeFalse(String message, **boolean** b) | Тест будет проигнорирован если параметр == true |
| assumeNotNull(Object... objects) | Тест будет проигнорирован если хотя бы один из переданных параметров == null |
| assumeNoException(Throwable t)  assumeNoException(String message, Throwable t) | Тест будет проигнорирован если параметр (исключение) не null |
| assumeThat(String message, T actual, Matcher<T> matcher)  assumeThat(T actual, Matcher<T> matcher) | Настраиваемое предположение, в качестве параметра передается org.hamcrest.Matcher – будет рассмотрен позже |

Предположения используются разработчиками очень редко (в отличии от утверждений) и как правило используются если тест зависит от окружения

Пример зависимости от окружения:

**public** **class** UnixClasspathExtractorTest {

**private** ClasspathExtractor extractor = **new** UnixClasspathExtractor();

@Test

**public** **void** testExtractClasspath() {

Assume.*assumeTrue*(System.*getProperty*("path.separator").equals(":"));

Assert.*assertArrayEquals*(extractor.extractCurrentUserPath(), **new** String[]{"/bin", "/usr/bin"});

}

}

Этот тест будет выполняться только в Unix-системах (например, на сервере Continuous Integration вроде Hudson или Jenkins).

Но все же желательно по возможности избегать привязки тестов к окружению и использовать кроссплатформенные библиотеки.