Contents

1. Intro to JUnit

2. Base components of JUnit

1 Intro to 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 -

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

2. Основные компоненты тестового класса

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

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.

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**

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

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

5)

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

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

При написании тестов на JUnit, необходимо учитывать как организована иерархия тестов в JUnit:

Набор Тестов 1 (Test Suite)

→ Набор Тестов 2 (Test Suite)

→ Тест-класс (Test Class)

→ Тест метод 1 (Test Method)

→ Тест метод 2 (Test Method)