# Лекция 13. Модульное тестирование с Junit

Модульное тестирование или unit testing — процесс проверки на корректность функционирования отдельных частей исходного кода программы путем запуска тестов в искусственной среде. Под частью кода в Java следует пони- мать исполняемый компонент.

С помощью модульного тестирования обычно тестируют низкоуровневые элементы кода — такие, как методы. JUnit позволяет вне исследуемого класса создавать тесты, при выполнении которых произойдет корректное завершение программы. Кроме основного положительного сценария может выполняться проверка работоспособности системы в альтернативных сценариях, например, при генерации методом исключения как реакция на ошибочные исходные данные.

Оценивая каждую часть кода изолированно и подтверждая корректность ее работы, точно установить проблему значительно проще, чем если бы элемент был частью системы. Технология позволяет и предлагает сделать более тесной связь между разработкой кода и его тестированием, а также предоставляет возможность проверить корректность работы класса, не прибегая к пробному выводу при отладке кода.

JUnit4 построен на аннотациях. Для использования технологии необходимо загрузить библиотеку JUnit с сервера junit.org и включить архив junit.jar в список библиотек приложения. При включении модульного тестирования в проект:

* тесты разрабатываются для нетривиальных методов системы;
* ошибки выявляются в процессе проектирования метода или класса;
* в первую очередь разрабатываются тесты на основной положительный сценарий;
* разработчику приходится больше уделять внимания альтернативным сценариям поведения, так как они являются источником ошибок, выявляемых на поздних стадиях разработки;
* разработчику приходится создавать более сфокусированные на своих обязанностях методы и классы, так как сложный код тестировать значительно труднее;
* снижается число новых ошибок при добавлении новой функциональности;
* устаревшие тесты можно игнорировать;
* тест отражает элементы технического задания, то есть некорректное завершение теста сообщает о нарушении технических требований заказчика;
* каждому техническому требованию соответствует тест;
* получение работоспособного кода с наименьшими затратами.

# Аннотация @Test

Аннотация ***@Test*** помечает метод как тестовый, что позволяет использовать возможности класса ***org.junit.Assert*** и запускать его в режиме тестирования. Тестовый метод должен всегда объявляться как ***public void***.

Аннотация может использовать параметры:

1. ***expected*** — определяет ожидаемый класс исключения;
2. ***timeout*** — определяет время, превышение которого делает тест ошибочным, применение которых будет рассмотрено ниже.

Пусть необходимо создать тест на метод, производящий простые вычисления студенческой стипендии.

|  |
| --- |
| **public** **interface** SalaryCalculator  {  **public** **double** calculateSalary(**double** coef);  }  **public** **class** SalaryCalculatorImpl **implements** SalaryCalculator  {  **public** **static** **final** **double** ***BASIC\_SALARY***=**100D**;  **public** **double** calculateSalary(**double** coef)  {  **return** ***BASIC\_SALARY*** \* coef;  }  } |
| Пример 1. Создание класса для проведения тестирования |

Метод, предназначенный для функционирования в качестве теста, достаточно пометить аннотацией ***@Test***. Простейший тест на метод будет выглядеть следующим образом.

|  |
| --- |
| **public** **class** SalaryCalculatorTest  {  @Test  **public** **void** scholarshipCalculateTest()  {  SalaryCalculator scholarshipCalculator=**new** SalaryCalculatorImpl();  **double** basicSalary=SalaryCalculatorImpl.***BASIC\_SALARY***;  **double** coef=**1.1D**;    **double** expected=basicSalary \* coef;  **double** actual=scholarshipCalculator.calculateSalary(coef);  //Assert.assertEquals(expected, actual, 0.00);  Assert.*assertEquals*("Тест не прошел, т.к.", expected, actual, **0.01**);  }  } |
| Пример 2. Создание теста для проверки работоспособности класса |

Метод ***assertEquals()*** проверяет на равенство значений ***expected*** и ***actual*** с возможной погрешностью ***delta***. При выполнении заданных условий сообщает об успешном завершении, в противном случае — об аварийном завершении теста. При аварийном завершении генерируется ошибка ***java.lang.AssertionError***. Все методы класса ***Assert*** в качестве возвращаемого значения имеют тип ***void***. Среди них можно выделить:

* ***assertTrue(boolean condition) / assertFalse(boolean condition)*** проверяет на истину/ложь значение condition;
* ***assertSame(Object expected, Object actual)*** проверяет, ссылаются ли ссылки на один и тот же объект;
* ***assertNotSame(Object unexpected, Object actual)*** проверяет, ссылаются ли ссылки на различные объекты;
* ***assertNull(Object object) / assertNotNull(Object object)*** проверяет, имеет или не имеет ссылка значение ***null***;
* ***assertThat(T actual, Matcher matcher)*** проверяет выполнение условия;
* ***fail()*** вызывает ошибку, используется для проверки, достигнута ли определенная часть кода или для заглушки, сообщающей, что тестовый метод пока не реализован.

Все перечисленные методы имеют перегруженную версию с параметром типа ***String*** в первой позиции, в который можно передавать сообщение, выводимое в случае аварийного завершения теста.

В дополнение к классу ***Assert*** с его методами жесткой проверки прохождения теста разработан класс ***org.junit.Assume***. Методы этого класса в случае невыполнения предполагаемого условия при работе теста сообщают только о том, что предположение не исполнилось, не генерируя при этом никаких ошибок. Методы класса предполагают, что:

* ***assumeNoException(Throwable t)*** тестируемый метод завершится, не вызвав исключения;
* ***assumeNotNull(Object…objects)*** передаваемый аргумент(ы) не является ссылкой на null;
* ***assumeThat(T actual, Matcher matcher)*** условие выполнится;
* ***assumeTrue(boolean b)*** значение передаваемого аргумента истинно.

# Фикстуры

***Фикстура (Fixture)*** — состояние среды тестирования, которое требуется для успешного выполнения тестового метода. Может быть представлено набором каких-либо объектов, состоянием базы данных, наличием определенных файлов, соединений и прочего.

В версии JUnit4 аннотации позволяют исполнять одну и ту же фикстуру для каждого теста или всего один раз для всего класса, или не исполнять ее совсем. Предусмотрено четыре аннотации фикстур: две для фикстур уровня класса и две для фикстур уровня метода:

1. ***@BeforeClass*** запускается только один раз при запуске теста.
2. ***@Before*** запускается перед каждым тестовым методом.
3. ***@After*** запускается после каждого метода.
4. ***@AfterClass*** запускается после того, как отработали все тестовые методы.

Использование фикстур позволяет выделить этапы и точно определять моменты, например, создания/удаления объекта, инициализации необходимых ресурсов, очистки памяти и проч.

Пусть метод ***getSalaryCoefficient(int averageMark)*** возвращает повышающий коэффициент стипендии в зависимости от среднего балла студента.

|  |
| --- |
| **public** **class** SalaryCalculatorImpl **implements** SalaryCalculator  {  …  **public** **double** getSalaryCoefficient(**int** averageMark)  {  **double** salaryCoefficient;  **switch** (averageMark)  {  **case** **3**:  salaryCoefficient=**1**;  **break**;  **case** **4**:  salaryCoefficient=**1.3**;  **break**;  **case** **5**:  salaryCoefficient=**1.5**;  **break**;  **default**:  salaryCoefficient=**0**;  }    **return** salaryCoefficient;  }  } |
| Пример 3. Добавление нового метода в класс SalaryCalculatorImpl |

|  |
| --- |
| **public** **class** SalaryCalculatorImplTest  {  **private** SalaryCalculatorImpl calculator;  @Before  **public** **void** initSalaryCalculator()  {  calculator=**new** SalaryCalculatorImpl();  }  @After  **public** **void** removeSalaryCalculator()  {  calculator=**null**;  }  @Test  **public** **void** testGetCoefficientForFive()  {  **double** expected=**1.5**;  **double** actual=calculator.getSalaryCoefficient(**5**);  *assertEquals*("Coef for mark 5 is wrong:", expected, actual, **0.01**);  }  @Test  **public** **void** testGetCoefficientForThree()  {  **double** expected=**1**;  **double** actual=calculator.getSalaryCoefficient(**3**);  *assertEquals*("Coef for mark 3 is wrong:", expected, actual, **0.01**);  }  } |
| Пример 4. Добавление нового тестового метода с использованием фикстур Before и After |

Фикстурами ***@Before*** и ***@After*** заданы моменты создания объекта класса ***SalaryCalculatorImpl*** перед каждым новым тестом и удаления после его завершения: метод ***initSalaryCalculator()*** выполняет инициализацию поля ***calculator*** перед запуском каждого тестового метода, а метод ***removeSalaryCalculator()*** очистит это поле после завершения работы каждого из тестовых методов.

Можно переписать тестовый класс ***SalaryCalculatorImplTest*** с использованием фикстур @BeforeClass и @AfterClass, чье использование позволит создавать только один экземпляр класса, на котором и будут выполнены оба теста, а по завершении всех тестов – удалить этот объект.

|  |
| --- |
| **public** **class** SalaryCalculatorImplTest2  {  **private** **static** SalaryCalculatorImpl *calculator*;  @BeforeClass  **public** **static** **void** initSalaryCalculator()  {  *calculator*=**new** SalaryCalculatorImpl();  }  @Test  **public** **void** testGetCoefficientForFive()  {  **double** expected=**1.5**;  **double** actual=*calculator*.getSalaryCoefficient(**5**);  System.***out***.println("testGetCoefficientForFive: calculator=" + *calculator*);  System.***out***.println("testGetCoefficientForFive: expected=" + expected + " and actual=" + actual);    *assertEquals*("Coef for mark 5 is wrong:", expected, actual, **0.01**);  }  @Test  **public** **void** testGetCoefficientForThree()  {  **double** expected=**1**;  **double** actual=*calculator*.getSalaryCoefficient(**3**);    System.***out***.println("testGetCoefficientForFive: calculator=" + *calculator*);  System.***out***.println("testGetCoefficientForThree: expected=" + expected + " and actual=" + actual);  *assertEquals*("Coef for mark 3 is wrong:", expected, actual, **0.01**);  }  } |
| Пример 5.Изменение тестового класса с использованием фикстур BeforeClass и AfterClass |

Метод ***initSalaryCalculator()*** отработает один раз перед вызовом первого тестового метода. Метод, отмеченный аннотацией ***@BeforeClass*** или ***@AfterClass***, должен быть ***статическим***. JUnit4 позволяет указать для тестового сценария более одной фикстуры. Новые фикстуры на основе аннотаций не препятствуют созданию нескольких фикстурных методов ***@BeforeClass*** и ***@AfterClass***. Однако их порядок исполнения задать нельзя.

# Тестирование исключительных ситуаций

При тестировании альтернативных сценариев работы метода часто требуется точно определить тип генерируемого методом исключения на основе переданных некорректных параметров. Если тест выдает исключение, то инфраструктура тестирования сообщает о корректном результате его исполнения. Аннотацию ***@Test*** при необходимости тестирования генерации конкретного исключения следует использовать с параметром ***expected***. Параметр предназначен для задания типа исключения, которое данный тест должен генерировать в про- цессе своего выполнения.

Пусть метод ***getSalaryCoefficient(int averageMark)*** генерирует исключение ***UnknownMarkException***, если число, переданное в метод, не входит в интервал возможных оценок.

|  |
| --- |
| **public** **class** SalaryCalculatorImpl **implements** SalaryCalculator  {  …  **public** **double** getSalaryCoefficient(**int** averageMark)  **throws** UnknownMarkException  {  **double** salaryCoefficient;  **switch** (averageMark)  {  **case** **3**:  salaryCoefficient=**1**;  **break**;  **case** **4**:  salaryCoefficient=**1.3**;  **break**;  **case** **5**:  salaryCoefficient=**1.5**;  **break**;  **default**:  **throw** **new** UnknownMarkException();  }    **return** salaryCoefficient;  }  } |
|  |
| **public** **class** SalaryCalculatorImplTest  {  …  **@Test(expected=UnknownMarkException.class)**  **public** **void** testGetCoefficientException()  **throws** UnknownMarkException  {  **double** expected=**1**;  **double** actual=calculator.getSalaryCoefficient(**100**);  System.***out***.println("testGetCoefficientForhree: calculator=" + calculator);  System.***out***.println("testGetCoefficientForThree: expected=" + expected + " and actual=" + actual);  *assertEquals*("For mark 100 wasn't exception: ", expected, actual, **0.01**);  }  } |
| Пример 6. Перехват ожидаемого исключения, генерируемого тестируемым методом |

Тест завершится успешно лишь в том случае, если возникнет исключительная ситуация. Исключение ***UnknownMarkException*** необходимо указать в секции ***throws*** тестового метода по правилам работы с проверяемыми исключениями. Может возникнуть необходимость проверить не только возникновение исключительной ситуации, но и текст сообщения в экземпляре исключения. В этом случае лучше прибегнуть к обычному подходу без параметра ***expected***.

# Ограничение по времени

В качестве параметра тестового сценария аннотации ***@Test*** может быть использовано значение лимита времени ***timeout***. Параметр ***timeout*** определяет максимальный временной промежуток в миллисекундах, отводимый на исполнение теста. Если выделенное время истекло, а тест продолжает выполняться, то тест завершается неудачей.

|  |
| --- |
| **public** **class** SalaryCalculatorImplTest  {  …    @Test(timeout=**1**)  **public** **void** testGetCoefficientTimeout()  {  **for** (**int** i=**1**;i < **100000**;i++)  {  **double** stepUpCoefficient=**1** / i;  **double** expected=**100** \* stepUpCoefficient;  **double** actual=calculator.calculateSalary(stepUpCoefficient);  *assertEquals*(expected, actual, **0.01**);  }  }  } |
| Пример 7. Использование временных ограничений для проведения теста |

# Игнорирование тестов

При контроле корректности функционарования бизнес-логики приложений п до появления JUnit4 игнорирование неудачных, незавершенных или устаревших тестов представляло определенную проблему. Аннотация ***@Ignore*** заставляет инфраструктуру тестирования проигнорировать данный тестовый метод. Аннотация предусматривает наличие комментария о причине игнорирования теста, полезного при следующем к нему обращении.

# Правила

Аннотация ***@Rule*** позволяет более гибко работать с классами утилитами, определяющими правила работы с тестами. Некоторые классы пакета ***org.junit.rules*** повторяют функциональность параметров аннотации ***@Test***, а именно, класс ***Timeout*** определяет таймаут для теста, класс ***ExpectedException*** — ожидаемое исключение.

С другой стороны, класс ***TemporaryFolder*** дает возможность протестировать возможность управления временными файлами и директориями.

Пусть в класс ***SalaryCalculatorImpl*** добавлена возможность печати отчета в файл в методе ***writeResult(File f)***. Тогда с использованием правил можно создать тест для метода записи в файл и два теста, проверяющих генерацию исключений.

|  |
| --- |
| **public** **class** SalaryCalculatorImpl **implements** SalaryCalculator  {  …  **public** **void** writeResult(File f) **throws** IOException  {  FileWriter fw=**new** FileWriter(f);  fw.append(**this**.toString());  fw.flush();  fw.close();  }  } |
| **public** **class** SalaryCalculatorImplTest  {  …  @Rule  **public** **final** TemporaryFolder folder=**new** TemporaryFolder();  @Test  **public** **void** writeResultTest() **throws** IOException  {  File file=folder.newFile("result.txt");  *calculator*.writeResult(file);  } } |
| Пример 8. Использование правил для проведения теста |

# Наборы тестов и параметризованные тесты

При контроле корректности функционарования бизнес-логики приложений приходится создавать тесты, число которых достаточно велико и непостоянно. Аннотация ***@RunWith*** задает способ запуска теста, в случае ***@RunWith(Suite.class)*** — запуск набора тестов, перечисляемых в аннотации ***@SuiteClasses***.

|  |
| --- |
| **import** org.junit.runner.RunWith;  **import** org.junit.runners.Suite;  **import** org.junit.runners.Suite.SuiteClasses;  @SuiteClasses(  {SalaryCalculatorImplTest.**class**, SalaryCalculatorImplTest2.**class** })  @RunWith(Suite.**class**)  **public** **class** SalaryCalculatorSuite  {  // nothing else  } |
| Пример 9. Автоматический запуск набора тестов, которые содержатся в классах, указанных в аннотации ***@SuiteClasses*** |

Наличие класса после описания способа группы тестов в общем случае необходимая формальность. JUnit4 позволяет создавать тест, который может работать с различными наборами значений параметров, что позволяет разработать единый тестовый сценарий и запускать его несколько раз — по числу наборов параметров.

Запуск параметризованного теста с набором данных также требует аннотации ***@RunWith(Parameterized.class)***, где в качестве значения передается указание на способ запуска. У класса должны быть поля по числу параметров в каждом наборе. Аннотация ***@Parameters*** маркирует статический метод, который возвращает данные для теста, представленные типом ***Collection***. Кроме того, необходим ***public***-конструктор, связывающий данные для теста и поля класса.

|  |
| --- |
| **import** org.junit.Test;  **import** org.junit.runner.RunWith;  **import** org.junit.runners.Parameterized;  **import** org.junit.runners.Parameterized.Parameters;  **import** salary.SalaryCalculatorImpl;  @RunWith(Parameterized.**class**)  **public** **class** SalaryCalculatorImplTest3  {  **private** **int** averageMark;  **private** **double** coef;  **public** SalaryCalculatorImplTest3(**int** averageMark, **double** coef)  {  **this**.averageMark=averageMark;  **this**.coef=coef;  }  @Parameters  **public** **static** List<Object[]> initParameters()  {  **return** Arrays.*asList*(**new** Object[][] {  { **3**, **1.0** }, { **4**, **1.3** }, { **5**, **1.5** } });  }  @Test  **public** **void** testGetCoefficient()  {  SalaryCalculatorImpl salaryCalculator=**new** SalaryCalculatorImpl();  System.***out***.println("coef=" +coef+", averageMark="+averageMark);  **double** expected=coef;  **double** actual =  salaryCalculator.getSalaryCoefficient(averageMark);  *assertEquals*(expected, actual, **0.01**);  }  } |
| Пример 10. Автоматический запуск теста, с разными значениями входных параметров |