Разработка тестов с помощью инструмента Summer

Оглавление

[Введение 2](#_Toc339463573)

[Основные понятия 2](#_Toc339463574)

[Примеры разработки тестов 3](#_Toc339463575)

[Простейшие тесты 3](#_Toc339463576)

[Параметризованные тесты 5](#_Toc339463577)

[Тесты, использующие состояния 6](#_Toc339463578)

[Параметризованные тесты, использующие состояния 12](#_Toc339463579)

[Тесты с использованием охранных условий 15](#_Toc339463580)

[Настроечные методы 17](#_Toc339463581)

[Компоновка теста с использованием групп методов 17](#_Toc339463582)

[Использование программных контрактов 17](#_Toc339463583)

[Использование заглушек 17](#_Toc339463584)

[Компоновка сложных тестов 17](#_Toc339463585)

[Справочник по основным конструкциям 18](#_Toc339463586)

[Параметры командной строки 18](#_Toc339463587)

[Описание компоновки выполняемых тестов 18](#_Toc339463588)

[Опции выбора тестовых методов 19](#_Toc339463589)

[Конфигурация теста 19](#_Toc339463590)

[Уровень трассировки 19](#_Toc339463591)

[Используемый алгоритм обхода автомата 20](#_Toc339463592)

[Ограничения на тестовую последовательность 21](#_Toc339463593)

[Ограничение на число обнаруженных ошибок 21](#_Toc339463594)

[Политика восстановления после ошибок 21](#_Toc339463595)

[Описание конфигурации теста 21](#_Toc339463596)

[Основные элементы теста 22](#_Toc339463597)

[Тестовый класс и тестовый метод 22](#_Toc339463598)

[Дескриптор состояния 22](#_Toc339463599)

[Источник данных 22](#_Toc339463600)

[Охранное условие 22](#_Toc339463601)

[Настроечные элементы теста 22](#_Toc339463602)

[Библиотечные классы 22](#_Toc339463603)

# Введение

Данный документ содержит описание основных принципов и техник разработки тестов с помощью инструмента Summer, разобранные примеры таких тестов и краткий справочник по основным конструкциям, используемым при создании тестов с помощью этого инструмента.

# Основные понятия

Разработка тестов с помощью инструмента Summer основана на следующих идеях.

* Для проверки корректности работы тестируемой ситемы (system under test, SUT) обычно требуется много тестов, оказывающих на нее различные воздействия в разных состояниях.

Организация тестовых воздействий в разных состояниях SUT в инструменте Summer сделана так: определяем, какие данные рассматриваются как состояние теста, какие воздействия можно применять в произвольном состоянии, а затем выполняем все заданные воздействия во всех состояниях теста, которые будут обнаружены во время его работы.  
Если выполнение различных воздействий не порождает новые состояния бесконечно, то набор достижимых состояний ограничен, и выполнение любого воздействия в любом состоянии снова приводит в одно из состояний этого набора. Конечный *автомат теста* имеет состояния, являющиеся достижимыми состояниями теста, а его переходы соответствуют выполнению воздействий в разных состояниях (само воздействие оказывается входным символом для этого автомата). Полное выполнение теста при этом соответствует такому пути по переходам в автомате теста, что каждый переход рано или поздно окажется пройденным. Тест заканчивается, когда все переходы пройдены, т.е. каждое воздействие выполнено в каждом обнаруженном состоянии.  
Реализованное в инструменте Summer построение тестов прямо во время выполнения, (до выполнения мы можем не знать, какие состояния окажутся достижимыми) называется *динамическим построением тестов* (online test generation, on-the-fly test generation).

* Для упрощения описания тестов используются те же идеи, что реализованы в инструментах модельного тестирования типа xUnit.
  + Сценарий выполнения каждого воздействия описывается в виде метода в Java-классе, помеченного аннотацией @Test и называемого *тестовым методом*.
  + Тестовые методы, для выполнения которых нужны одни и те же общие данные, общие вспомогательные методы и пр., группируются в тестовые классы. *Тестовый класс* является обычным Java-классом, помеченным аннотацией @Test.
  + Для задания состояния теста используются методы-дескрипторы состояния. *Метод-дескриптор состояния* представляет собой метод тестового класса, помеченным аннотацией @State.

Стоит отметить, что, в отличие от инструментов типа xUnit, в которых тестируемая система ре-инициализируется после выполнения каждого тестового метода, в Summer этого не происходит — иначе отслеживание состояний теста было бы не нужно и невозможно было бы выполнить одно воздействие в разных состояниях.

* Имеется набор дополнительных возможностей, позволяющих строить более сложные тесты, определять параметризованные тестовые воздействия, выполнять какие-либо вспомогательные действия непосредственно перед или после каждого метода, определять условия выполнения данного тестового метода, и т.д.
* Проверяемые ограничения на реакции SUT в зависимости от выполняемых тестовых воздействий можно описывать отдельно от тестов в виде *программных контрактов*, разделяя тем самым логику выполняемых тестом проверок и построение различных ситуаций, в которых нужно проверить поведение SUT.  
  Но можно и не выделять проверки в отдельные контракты, а описывть их в самом коде тестовых методов.

Использование автоматной модели для построения тестов (а также возможность использования контрактов) означает, что используемые в Summer техники тестирования относятся к *тестированию на основе моделей* (model based testing).

# Примеры разработки тестов

В данном разделе описываются разнообразные примеры разработки тестов с помощью Summer, иллюстрирующие основные возможности инструмента.

## Простейшие тесты

Простейшие тесты с помощью Summer можно создавать и без использования механизма обхода автомата — для этого достаточно не указывать в тестовом классе дескриптор состояния. Получаемые тесты при этом очень похожи на тесты, создаваемые в инструментах типа xUnit, за исключением того, что тестируемые объекты не создаются заново после выполнения каждого тестового метода.

Ниже приведен пример такого теста для функции извлечения квадратного корня.

**import** mbtest.annotations.Test;

**import** mbtest.contracts.Contract;

@Test

**public** **class** SqrtTest

{

@Test

**public** **void** testZero()

{

Contract.*assertEquals*(Math.*sqrt*(0.0), 0.0, "Square root of 0 should be 0");

}

@Test

**public** **void** testOne()

{

Contract.*assertEquals*(Math.*sqrt*(1.0), 1.0, "Square root of 1 should be 1");

}

@Test

**public** **void** testFour()

{

Contract.*assertEquals*(Math.*sqrt*(4.0), 2.0, "Square root of 4 should be 2");

}

@Test

**public** **void** testNine()

{

Contract.*assertEquals*(Math.*sqrt*(9.0), 3.0, "Square root of 9 should be 3");

}

}

В данном примере тестовый класс SqrtTest помечен аннотацией @Test, так же как и четыре тестовых метода. Для проверок используется вспомогательный метод Contract.assertEquals(), имеющий три параметра: полученный результат, ожидаемое значение результата и сообщение, выдаваемое в трассу при расхождении между ними.

Для выполнения этого теста нужно запустить класс mbtest.MBTester, например, так (помимо указанных опций нужно правильно указать classpath для Java-машины):

java mbtest.MBTester -testclass mbtest.tests.guide.sqrt.SqrtTest –explorer none -ignorefails

Другой способ — сделать в Eclipse специальный класс-стартер следующего вида.

**import** mbtest.MBTester;

**public** **class** Runner

{

**public** **static** **void** main(String[] args)

{

MBTester.*main*(**new** String[] {

"-testclass" , "mbtest.tests.guide.sqrt.SqrtTest"

,"-explorer" , "none"

,"-ignorefails"

});

}

}

Этот класс можно запускать кнопкой Launch в Eclipse.

В полученной после запуска теста трассе будут сообщения об успешном выполнении всех 4-х тестовых методов.

Чтобы просимулировать ошибку, исправим код последнего метода таким образом.

@Test

**public** **void** testNine()

{

Contract.*assertEquals*(Math.*sqrt*(9.0), 3.1, "Square root of 9 should be 3");

}

Теперь ожидается, что квадратный корень из 9 вернет значение 3.1. Выполнив исправленный тест, получим следующую трассу.

INFO : TestRunner: Starting test Command line test

INFO : Explorer: Current state: <Single> is new

INFO : Explorer: Current number of states = 1, number of transitions = 0

INFO : Explorer: Executing transition public void mbtest.tests.guide.SqrtTest.testZero() [new]

INFO : Explorer: Executing transition public void mbtest.tests.guide.SqrtTest.testFour() [new]

INFO : Explorer: Executing transition public void mbtest.tests.guide.SqrtTest.testNine() [new]

ERROR : Contract: Assertion failed: Square root of 9 should be 3: actual 3.0 isn't equal to expected 3.1

INFO : Explorer: Ignoring failure : current number of failures is 1, while 20 is allowed

INFO : Explorer: Executing transition public void mbtest.tests.guide.SqrtTest.testOne() [new]

SUMMARY: Explorer: All is tested

SUMMARY: Explorer: Total number of failures = 1

SUMMARY: Explorer: Total number of states = 1

SUMMARY: Explorer: Total number of transitions = 4

SUMMARY: Explorer: Total path length = 4

SUMMARY: Explorer: Total time = 0

В ней содержится сообщение об ошибке при выполнении четвертого тестового метода. При этом показаны как ожидаемое, так и полученное из тестируемой системы значения.

Можно убрать из трассы информационные сообщения, оставив только сообщение об ошибке и итоговую информацию о выполнении теста (summary). Для этого используется опция -loglevel summary.

## Параметризованные тесты

За счет параметризации можно сделать тест для квадратного корня более представительным. При этом, однако, нужно реализовать процедуру проверки корректности результата таким образом, чтобы она подходила для всех используемых значений. Один из способов сделать это представлен ниже.

**import** mbtest.annotations.DataProvider;

**import** mbtest.annotations.Test;

**import** mbtest.contracts.Contract;

@Test

**public** **class** PSqrtTest

{

@Test

@DataProvider(name="params")

**public** **void** testAny(**double** x)

{

**double** r = Math.*sqrt*(x);

Contract.*assertEquals*(r\*r, x

, "Square root squared should be equal to the argument");

}

**double**[] params = **new** **double**[]{0., 1., 4., 9., 16., 25., 100., 169., 225.};

}

В приведенном примере единственный тестовый метод имеет один параметр, значение которого в нем используется как аргумент вызова квадратного корня. Значения для параметра задаются при помощи аннотации @DataProvider, которая указывает *исчтоник данных* — в данном случае поле, содержащего коллекцию значений, в данном случае это массив из 9-ти чисел.

Источником данных может служить метод, возвращающий коллекцию значений или итератор по ней. С помощью итератора можно, например, реализовать чтение данных из файла.

Для проверки правильности результата в приведенном примере используется возведение полученного квадратного корня в квадрат. Эта процедура вполне подходит для проверки вычислений квадратного корня из квадратов небольших целых чисел, однако, в силу дискретности чисел типа double, не работает для произвольного значения этого типа. Для устранения этого недостатка можно использовать еще один параметр — правильное значение корня, — и заранее вычисленные корректные значения для него.

После внесения соответствующих изменений тестовый класс будет выглядеть следующим образом.

@Test

**public** **class** PVSqrtTest

{

@Test

@DataProvider(name="params")

**public** **void** testAny(**double** x, **double** v)

{

Contract.*assertEquals*(Math.*sqrt*(x), v

, "Square root of " + x + " should be equal to " + v);

}

Object[][] params = **new** Object[][]{

{0. , 0.}, { 1., 1.}, { 4., 2.}, { 9., 3.}, { 16., 4.}

, {25., 5.}, {100., 10.}, {169., 13.}, {225., 15.}, {289., 17.}

, {2, 1.41421356237309505}, {3., 1.7320508075688773}

};

}

При использовании одного источника данных для нескольких параметров этот источник данных должен быть коллекцией массивов типа Object[], или возвращать такую коллекцию, или давать итератор по такой коллекции.

## Тесты, использующие состояния

В качестве примера теста, использующего механизм обхода автомата, создадим тест для простой реализации ограниченного стека. Код этой реализации приведен ниже.

**public** **class** SmallStack

{

**public** **final** **static** **int** *MAX\_SIZE* = 5;

Object[] items = **new** Object[*MAX\_SIZE*];

**int** size = 0;

**public** **int** size() { **return** size; }

**public** **boolean** push(Object o)

{

**if**(size == *MAX\_SIZE*) **return** **false**;

**else**

{

items[size++] = o;

**return** **true**;

}

}

**public** Object pop()

{

**if**(size == 0) **return** **null**;

**else** **return** items[--size];

}

**public** Object head()

{

**if**(size == 0) **return** **null**;

**else** **return** items[size-1];

}

}

Этот класс имеет 4 метода.

* Метод size() возвращает количество элементов в стеке.
* Метод pop() для непустого стека возвращает последний добавленный элемент и удаляет его. Если стек пуст, этот метод возвращает null.
* Метод head() возвращает последний добавленный элемент для непустого стека (не удаляя его) и null для пустого.
* Метод push() добавляет свой аргумент в стек, если тот не заполнен полностью. Полный стек этот метод не меняет. Максимально возможное количество элементов в стеке в данной реализации равно 5.

Тест для ограниченного стека выглядит так.

**import** java.util.List;

**import** java.util.LinkedList;

**import** mbtest.annotations.State;

**import** mbtest.annotations.Test;

**import** mbtest.contracts.Contract;

@Test

**public** **class** StackTest

{

SmallStack stack = **new** SmallStack();

List<Object> list = **new** LinkedList<Object>();

@State

**public** **int** getSize() { **return** stack.size(); }

@Test

**public** **void** testPush()

{

**int** oldSize = stack.size();

Object o = **new** Object();

**boolean** res = stack.push(o);

**if**(oldSize < SmallStack.*MAX\_SIZE*)

{

Contract.*assertTrue*(res, "The result should be true");

Contract.*assertEquals*(stack.size(), oldSize+1

, "Size should increase by 1");

Contract.*assertIdentical*(stack.head(), o

, "The object pushed should become the head");

list.add(o);

}

**else**

{

Contract.*assertFalse*(res, "The result should be false");

Contract.*assertEquals*(stack.size(), oldSize

, "Size should not be changed");

Contract.*assertIdentical*(stack.head(), list.get(oldSize-1)

, "The head should not be changed");

}

}

@Test

**public** **void** testPop()

{

**int** oldSize = stack.size();

Object res = stack.pop();

**if**(oldSize > 0)

{

Contract.*assertEquals*(stack.size(), oldSize-1

, "Size should decrease by 1");

Contract.*assertIdentical*(res, list.get(oldSize-1)

, "The result should be the old head");

list.remove(oldSize-1);

}

**else**

Contract.*assertEquals*(stack.size(), oldSize

, "Size should not be changed");

}

@Test

**public** **void** testHead()

{

**int** oldSize = stack.size();

Contract.*assertEquals*(stack.size(), oldSize, "Size should not be changed");

**if**(oldSize > 0)

Contract.*assertIdentical*(stack.head(), list.get(oldSize-1)

, "The head should not be changed");

**else**

Contract.*assertIdentical*(stack.head(), **null**, "The head should be null");

}

@Test

**public** **void** testSize()

{

Contract.*assertEquals*(stack.size(), list.size()

, "Size should be equal to the size of stored list");

Contract.*assertEquals*(stack.size(), list.size()

, "size() call should not change the size");

}

}

Этот тест имеет следующую структуру.

* Одно из полей (stack) является ссылкой на тестируемый объект. Это поле инициализируется при инициализации теста.
* Состоянием теста считается размер тестируемого стека (метод getSize()).
* Для каждого общедоступного метода проверяемого класса определен тестовый метод, который описывает проверку работы первого метода в общей ситуации, независимо от текущего состояния.  
  Такая структура теста обеспечивает, что при его выполнении с помощью обхода автомата в каждом обнаруженном состоянии будет вызван и проверен каждый общедоступный метод тестируемого объекта.
* Чтобы выполнить такую проверку часто необходимо независимо хранить какие-то данные о состоянии тестируемой системы, которые позволяют полностью проверять результаты работы всех методов. В данном примере такими данными является список текущих элементов стека list (удаленные элементы уже не хранятся — они не влияют на результаты последующих обращений к методам стека) — он позволяет проверить, что очередной результат метода pop() действительно совпадает с когда-то добавленным методом push() объектом, и, более того, проверить, что этот объект появляется в полном соответствии с реализуемой стеком логикой FIFO, не раньше и не позже.  
  Такие вспомогательные данные, которые позволяют проверить корректность результатов вызова произвольного метода в произвольном состоянии, и которые нужно обновлять при модификации состояния тестируемой системы (заметьте, что list обновляется при вызовах push() в неполном стеке и вызовах pop() в непустом), называются *модельным состоянием* тестируемой системы.
* Проверки для методов, не изменяющих стек, написаны, в основном, в терминах данных модельного состояния.
* В каждом тестовом методе написаны проверки изменения или сохранения для всех непосредственно доступных элементов стека (размера и головы стека). При этом используются методы, которые не должны менять стек.  
  Возможны достаточно хитрые ошибки в работе методов стека, которые могут быть не обнаружены во время таких проверок сразу. При этом, однако, очень трудно придумать ошибочную реализацию стека, которая работала бы с ошибками, не заметными для такого теста во всех его достижимых состояниях (т.е. для всех возможных размеров стека). Таким образом, даже если ошибка не будет обнаружена сразу, она приведет к рассогласованию моельного состояния и состояния тестируемого объекта, которое с большой вероятностью будет замечено далее.  
  Альтернативный способ исключить такие ошибки (способный, однако, еще более отдалить момент возникновения ошибки в SUT от момента ее обнаружения) — вообще не полагаться на результаты работы других методов, кроме проверяемого, использовать в проверках только данные модельного состояния и результаты работы проверяемого метода.

Помимо метода Contract.assertEquals(actual, expected, msg) в проверках используются методы Contract.assertTrue(actual, msg) — эквивалентен Contract.assertEquals(actual, true, msg); Contract.assertFalse(actual, msg) — эквивалентен Contract.assertEquals(actual, false, msg); Contract.assertIdentical(actual, expected, msg) — проверяет идентичность, а не равенство объектов, т.е., что actual — тот же самый объект, что и expected, а не просто равный ему.

Отметим еще, что для представления модельного состояния выбран LinkedList, а стек реализован на основе массива. Это сделано так, чтобы логика работы коллекций, лежащих в основе проверяемой реализации и модели, была достаточно различной, и, тем самым, снижался риск одинаковых ошибок. Наличие одинаковой ошибки в работе проверяемой системы и работе с модельным состоянием может привести к тому, что тест не обнаружит ее, несмотря на тщательность самого тестирования. Если в проверяемой системе и при работе с модельным состоянием в тесте одним и тем же образом используются одинаковые объекты-коллекции, то тест проверяет лишь, что два таких объекта одинаково реагируют на одни и те же вызовы, что не соответсвует его задачам.

При выполнении этого теста нужно указать значение опции -explorer, отличное от none, например, bfs (или не указывать никакого — по умолчанию используется подходящее значение). Эта опция в данном случае указывает, что необходимо строить обход автомата, а ее значение указывает используемый алгоритм построения его обхода (bfs — обход в ширину).

После выполнения этого теста получается трасса такого вида.

INFO : TestRunner: Starting test Command line test

INFO : Explorer: Current state: 0 is new

INFO : Explorer: Current number of states = 1, number of transitions = 0

INFO : Explorer: Executing transition public void mbtest.tests.guide.StackTest.testHead() [new]

INFO : Explorer: Current state: 0 is old

INFO : Explorer: Executing transition public void mbtest.tests.guide.StackTest.testPush() [new]

INFO : Explorer: Current state: 1 is new

INFO : Explorer: Current number of states = 2, number of transitions = 2

INFO : Explorer: Executing transition public void mbtest.tests.guide.StackTest.testHead() [new]

INFO : Explorer: Current state: 1 is old

INFO : Explorer: Executing transition public void mbtest.tests.guide.StackTest.testPush() [new]

INFO : Explorer: Current state: 2 is new

INFO : Explorer: Current number of states = 3, number of transitions = 4

INFO : Explorer: Executing transition public void mbtest.tests.guide.StackTest.testHead() [new]

INFO : Explorer: Current state: 2 is old

INFO : Explorer: Executing transition public void mbtest.tests.guide.StackTest.testPush() [new]

INFO : Explorer: Current state: 3 is new

INFO : Explorer: Current number of states = 4, number of transitions = 6

INFO : Explorer: Executing transition public void mbtest.tests.guide.StackTest.testHead() [new]

INFO : Explorer: Current state: 3 is old

INFO : Explorer: Executing transition public void mbtest.tests.guide.StackTest.testPush() [new]

INFO : Explorer: Current state: 4 is new

INFO : Explorer: Current number of states = 5, number of transitions = 8

INFO : Explorer: Executing transition public void mbtest.tests.guide.StackTest.testHead() [new]

INFO : Explorer: Current state: 4 is old

INFO : Explorer: Executing transition public void mbtest.tests.guide.StackTest.testPush() [new]

INFO : Explorer: Current state: 5 is new

INFO : Explorer: Current number of states = 6, number of transitions = 10

INFO : Explorer: Executing transition public void mbtest.tests.guide.StackTest.testHead() [new]

INFO : Explorer: Current state: 5 is old

INFO : Explorer: Executing transition public void mbtest.tests.guide.StackTest.testPush() [new]

INFO : Explorer: Current state: 5 is old

INFO : Explorer: Executing transition public void mbtest.tests.guide.StackTest.testSize() [new]

INFO : Explorer: Current state: 5 is old

INFO : Explorer: Executing transition public void mbtest.tests.guide.StackTest.testPop() [new]

INFO : Explorer: Current state: 4 is old

INFO : Explorer: Executing transition public void mbtest.tests.guide.StackTest.testSize() [new]

INFO : Explorer: Current state: 4 is old

INFO : Explorer: Executing transition public void mbtest.tests.guide.StackTest.testPop() [new]

INFO : Explorer: Current state: 3 is old

INFO : Explorer: Executing transition public void mbtest.tests.guide.StackTest.testSize() [new]

INFO : Explorer: Current state: 3 is old

INFO : Explorer: Executing transition public void mbtest.tests.guide.StackTest.testPop() [new]

INFO : Explorer: Current state: 2 is old

INFO : Explorer: Executing transition public void mbtest.tests.guide.StackTest.testSize() [new]

INFO : Explorer: Current state: 2 is old

INFO : Explorer: Executing transition public void mbtest.tests.guide.StackTest.testPop() [new]

INFO : Explorer: Current state: 1 is old

INFO : Explorer: Executing transition public void mbtest.tests.guide.StackTest.testSize() [new]

INFO : Explorer: Current state: 1 is old

INFO : Explorer: Executing transition public void mbtest.tests.guide.StackTest.testPop() [new]

INFO : Explorer: Current state: 0 is old

INFO : Explorer: Executing transition public void mbtest.tests.guide.StackTest.testSize() [new]

INFO : Explorer: Current state: 0 is old

INFO : Explorer: Executing transition public void mbtest.tests.guide.StackTest.testPop() [new]

INFO : Explorer: Current state: 0 is old

INFO : Explorer: All states are tested

SUMMARY: Explorer: All is tested

SUMMARY: Explorer: Total number of failures = 0

SUMMARY: Explorer: Total number of states = 6

SUMMARY: Explorer: Total number of transitions = 24

SUMMARY: Explorer: Total path length = 24

SUMMARY: Explorer: Total time = 16

Числа, идентифицирующие состояния, соответствуют числу элементов в стеке. Всего было найдено 6 состояний (от пустого стека — 0 — до полного — 5) и 24 перехода (по 4 перехода, соответствующих тестовым методам, в каждом состоянии).

Можно восстановить порядок прохода по переходам на следующей диаграмме.

push

push

push

push

push

push, head, size

pop

pop

pop

pop

pop

pop, head, size

head, size

head, size

head, size

head, size

Эта диаграмма показывает состояния и переходы построенного при выполнении теста автомата. Если обозначить 0-push переход из состояния 0 по вызову метода push(), то представленная выше трасса соответствует следующей последовательности переходов.

0-head; 0-push; 1-head; 1-push; 2-head; 2-push; 3-head; 3-push; 4-head; 4-push; 5-head; 5-push; 5-size; 5-pop; 4-size; 4-pop; 3-size; 3-pop; 2-size; 2-pop; 0-size; 0-pop.

Легко видеть, что эта последовательность действительно покрывает все переходы в автомате.

При выполнении теста порядок выполнения отдельных переходов может быть различным (и, соответственно, полученные трассы могут различаться). В данном примере видно, что порядок первого выполнения тестовых методов в каждом состоянии дается последовательностью head; push; size; pop (в этом примере каждый переход выполняется один раз, поэтому все выполнения являются первыми, обычно же обход графа переходов требует повторного выполнения некоторых из них, которое делается уже не в заданном порядке, а в соответствии с логикой алгоритма построения обхода). При инициализации теста, даже одного и того же тестового класса при разных его выполнениях, эта последовательность может быть выстроена разными способами.

Чтобы обеспечить повторяемость теста с точность до порядка выполненя переходов (когда это необходимо), можно использовать *зависимости* между тестовыми методами. Такие зависимости определяют относительный порядок первого выполнения методов в каждом состоянии и описываются с помощью элементов аннотации @Test. В данном примере, чтобы зафиксировать именно приведенный порядок первых обращений к тестовым методам, достаточно переписать декларации этих методов следующим образом.

@Test(dependsOnMethods="testHead")

**public** **void** testPush() { ... }

@Test(dependsOnMethods="testSize")

**public** **void** testPop() { ... }

@Test

**public** **void** testHead() { ... }

@Test(dependsOnMethods="testPush")

**public** **void** testSize() { ... }

При отсутствии зависимостей порядок первого выполнения методов в состоянии выбирается недетерминированно при каждой инициализации теста, порядок выполнения методов в разных состояниях совпадает. При наличии зависимостей аналогичным образом выбирается один из порядков, при которых зависимый метод выполняется в данном состоянии после всех методов, от которых он зависит. Этот порядок далее используется для первых обращений к тестовым методам в каждом состоянии теста. Если в системе зависимостей есть цикл, при инициализации теста выдается сообщение об ошибке.

## Параметризованные тесты, использующие состояния

Рассмотрим пример теста для класса, реализующего множество целых чисел. Реализация построена на основе красно-черного сбалансированного дерева и реализует основные операции над множеством целых чисел.

* int size() — возвращает число элементов в множестве;
* boolean contains(int i) — проверяет, принадлежит ли заданное число множеству;
* boolean add(int i) — добавляет заданное число в множество, возвращает true, если такого элемента в множестве еще не было, и false иначе;
* boolean remove(int i) — удалается заданное число из множества, возвращает true, если такой элемент в множестве был, и false иначе.

Тест для множества целых чисел может выглядеть так.

**import** java.util.HashSet;

**import** mbtest.annotations.State;

**import** mbtest.annotations.Test;

**import** mbtest.annotations.DataProvider;

**import** mbtest.contracts.Contract;

@Test

**public** **class** IntSetTest

{

IntSet s = **new** IntSet();

HashSet<Integer> model = **new** HashSet<Integer>();

**int** n;

**public** **int**[] values;

**public** IntSetTest() { **this**(3); }

**public** IntSetTest(**int** n)

{

**this**.n = n;

values = **new** **int**[n];

**for**(**int** i = 0; i < n; i++)

{

values[i] = ((i&1) != 0)?((i+1)/2):(-i/2);

}

}

@State

@DataProvider(name = "values")

**public** **boolean** in(**int** i)

{

**return** model.contains(i);

}

@Test

**public** **void** size()

{

Contract.*assertEquals*(s.size(), model.size()  
 , "size() result should coincide with model");

}

@Test(dependsOnMethods="size")

@DataProvider(name = "values")

**public** **void** contains(**int** i)

{

Contract.*assertEquals*(s.contains(i), model.contains(i)  
 , "contains() result should coincide with model");

}

@Test(dependsOnMethods="remove")

@DataProvider(name = "values")

**public** **void** add(**int** i)

{

Contract.*assertEquals*(s.add(i), model.add(i)  
 , "add() result should coincide with model");

}

@Test(dependsOnMethods="contains")

@DataProvider(name = "values")

**public** **void** remove(**int** i)

{

Contract.*assertEquals*(s.remove(i), model.remove(i)

, "remove() result should coincide with model");

}

}

Этот тест имеет, в целом, ту же структуру, что и разобранный в предыдущем разделе пример, за исключением ряда особенностей.

* Зависимости между тестовыми методами необязательны — они определены только для фиксации порядка первых обращений.
* Введено модельное состояние model, реализованное с помощью HashSet, и все тестовые методы просто проверяют, что тестирумеый объект на все воздействия реагирует так же, как модельный. Поскольку тестируемая реализация основана на сбалансированном дереве, а реализация модельного объекта — на хэшировании, такой подход вполне оправдан.
* Инициализация теста сделана не вполне тривиально.  
  Введен параметр теста n (параметр конструктора тестового класса), который означает максимально возможное число элементов в множестве во время работы теста.  
  Объявлен массив целых чисел values, который инициализируется так, чтобы иметь n элементов, совпадающих с первыми n числами в следующей последовательности: 0, 1, -1, 2, -2, 3, -3, и т.д.
* Все тестовые методы, кроме size(), параметризованы и используют в качестве значений своих параметров элементы массива values.  
  Вычисление значений для параметров тестовых методов выполняется отдельно в каждом состоянии. При первом попадании в это состояние вычисляются и запоминаются итераторы, используемые как источники данных для параметризованных тестовых методов, или итераторы коллекций, являющихся такими источниками. При всех следующих попаданиях в этот состояние эти итераторы только используются. Таким образом, набор значений, используемых как аргументы тестовых методов, может зависеть от состояния, но определяется один раз — при первом попадании в этот состояние.  
  Порядок первых обращений к тестовому методу с параметром соответствует порядку, в котором источник данных отдает значения параметра.
* Дескриптор состояния тоже сделан параметризованным, с тем же массивом values в качестве источника данных.  
  Это означает, что реальным состоянием теста считается отображение элементов этого массива в те значения, которые выдает для них данный метод.  
  В нашем примере этот метод возвращает true или false для числа, содержащегося в values, в зависимости от того, содержится ли это число в множестве model или нет. Такое отображение чисел в {true, false} однозачно соответсвует множеству чисел, для которых оно дает true, т.е. состоянием данного теста, по сути, является копия множества model.

Сделать сложный объект состоянием теста можно и более прямым образом, например, переписать приведенный выше дескриптор состояния так.

@State

**public** Object state()

{

**return** ((HashSet<Integer>)model.clone());

}

Отметим, что поправленный таким образом дескриптор состояния возвращает не сссылку на model, а копию этого объекта. Это важно, поскольку алгоритм обхода запоминает полученные состояния, чтобы в дальнейшем определять, попал ли он в новое состояние или в уже пройденное. В этом случае сам объект model не годится, поскольку он изменяется в ходе работы теста. Возвращаемые дескриптором состояния объекты не должны меняться при дальнейшем выполнении теста, иначе результаты их сравнения друг с другом будут неадекватно отражать его текущее состояние. Это правило очень важно помнить при создании дескрипторов состояния — из-за его нарушения могут возникнуть трудно диагностирумые ошибки, которые внешне выглядят как совершенно непонятное поведение теста.

Помимо этого требования, объекты, возвращаемые дескриптором состояния теста должны иметь корректно определенные методы boolean equals(Object o) и int hashCode(). Первый метод нужен для сравнения состояний, второй используется для хранения объектов, представляющих состояния, в алгоритме обхода.

При использовании параметризованных дескрипторов состояния заботиться нужно, соответственно, о методах equals() и hashCode() для результатов методов-дескрипторов и значений, поставляемых источником данных, за исключением ситуаций, где они имеют примитивные типы.

Выполнить приведенный выше тест можно так же, как и описанный в предыдущем разделе. При этом получается следующая трасса (опция -loglevel выставлена в summary, рекомендуется один раз выставить ее в значение info и проанализировать последовательность выполняемых тестом действий).

SUMMARY: Explorer: All is tested

SUMMARY: Explorer: Total number of failures = 0

SUMMARY: Explorer: Total number of states = 8

SUMMARY: Explorer: Total number of transitions = 80

SUMMARY: Explorer: Total path length = 88

SUMMARY: Explorer: Total time = 94

Число состояний — 8 — соответствует числу различных множеств, элементами которых могут быть 0, 1, -1. Число переходов можно подсчитать так: в каждом состоянии по одному разу вызывается size(), а все остальные методы — contains(), add(), remove() — по 3 раза, итого, получаем 8+8\*3\*3 = 80 переходов.

Тестовый класс имеет конструктор с параметром (который имеет стандартное строковое представление), поэтому его можно выполнить с разными значениями этого параметра. Чтобы следать это, нужно указать в качестве значения опции -testclass полное имя тестового класса вместе с нужным значением параметра, например, так: mbtest.tests.guide.IntSetTest(7). При выполнении теста с таким значением и опцией -loglevel, равной summary, получается такая трасса.

SUMMARY: Explorer: All is tested

SUMMARY: Explorer: Total number of failures = 0

SUMMARY: Explorer: Total number of states = 128

SUMMARY: Explorer: Total number of transitions = 2816

SUMMARY: Explorer: Total path length = 2968

SUMMARY: Explorer: Total time = 172

Здесь обнаруженное число состояний равно 27 = 128, а число переходов — 27+27\*3\*7 = 2816.

## Тесты с охранными условиями

Иногда не во всех состояниях можно вызывать все тестовые методы. Такая ситуация может возникнуть по двум основным причинам: либо какой-то метод нельзя вызывать в каких-то состояниях тестируемого объекта, потому что поведение его в таком случае не определено (например, нельзя читать из закрытого файла), либо для обеспечения конечности теста имеет смысл запретить в состояниях, далеких от начального, вызов методов, уводящих от начального состояния еще дальше (при этом поведение этих методов достаточно тщательно проверяется в других состояниях и имеется очень незначительный риск того, что ошибки в их поведении возникают только в далеких состояниях).

Ниже приведен пример теста для списка, в котором возможность выполнения тестовых методов для добавления новых элементов ограничена по второй причине.

**import** java.util.List;

**import** java.util.ArrayList;

**import** java.util.LinkedList;

**import** mbtest.annotations.Test;

**import** mbtest.annotations.State;

**import** mbtest.annotations.Guard;

**import** mbtest.annotations.DataProvider;

**import** mbtest.contracts.Contract;

@Test

**public** **class** ListTest

{

List<Integer> list = **new** ArrayList<Integer>();

List<Integer> model = **new** LinkedList<Integer>();

**protected** **int** maxValue;

**protected** **int** maxLength;

**int**[] values;

**int**[] indeces()

{

**int** s = model.size();

**int**[] r = **new** **int**[s];

**for**(**int** i = 0; i < s; i++) r[i] = i;

**return** r;

}

**boolean** sizeBound() { **return** model.size() < maxLength; }

**public** ListTest()

{

**this**(2, 2);

}

**public** ListTest(**int** maxValue, **int** maxLength)

{

**this**.maxValue = maxValue;

**this**.maxLength = maxLength;

values = **new** **int**[maxValue];

**for**(**int** i = 0; i < maxValue; i++) values[i] = i;

}

@State

**public** Object state() { **return** ((LinkedList<Integer>)model).clone(); }

@Test

@DataProvider(name="indeces")

**public** **void** remove(**int** i)

{

Contract.*assertEquals*(list.remove(i), model.remove(i)

, "remove() result should be the same as for model");

Contract.*assertEquals*(list, model

, "Contents after remove() should be the same as for model");

}

@Test

@DataProvider(name="indeces")

**public** **void** get(**int** i)

{

Contract.*assertEquals*(list.get(i), model.get(i)

, "get() result should be the same as for model");

Contract.*assertEquals*(list, model

, "Contents after get() should be the same as for model");

}

@Test

@Guard(names = "sizeBound")

**public** **void** addInternal(

@DataProvider(name="indeces") **int** i, @DataProvider(name="values") **int** j)

{

list.add(i, j);

model.add(i, j);

Contract.*assertEquals*(list, model

, "Contents after internal add() should be the same as for model");

}

@Test

@Guard(names = "sizeBound")

@DataProvider(name="values")

**public** **void** addLast(**int** i)

{

list.add(model.size(), i);

model.add(model.size(), i);

Contract.*assertEquals*(list, model

, "Contents after add() in the end should be the same as for model");

}

}

Этот тест имеет, в целом, ту же структуру, что и разобранные ранее. Особенности ее таковы.

* У теста два параметра — максимальная длина тестируемого списка и максимальное возможное значение элемента (в списке в ходе теста могут встречаться элементы от 0 до этого максимума).
* Источник данных indeces() реализован в виде метода, который возвращает массив, содержащий целые числа от 0 до текущей длины списка, исключая последнюю.
* Источники данных для параметров тестового метода addInternal() указаны по отдельности. В таком случае в качестве возможных наборов значений параметров фигурируют все возможные пары значений, где первое взято из первого источника, второе — из второго.  
  Если параметров больше двух, то, соответственно, будут строиться все возможные комбинации значений из указанных источников данных.
* Тестовые методы addInternal() и addLast() имеют *охранные условия*, задаваемые при помощи аннотации @Guard с элементом names, указывающим имя метода, который должен вернуть true, чтобы соответсвующий тестовый метод можно было выполнить.  
  В элементе names можно указать несколько имен методов. Все они должны возвращать булевский результат и либо не иметь параметров (и тем самым, зависеть только от текущего состояния), либо иметь такие же параметры, как и метод с таким охранным условием (в этом случае для одних наборов аргументов условие может быть выполнено, для других — нет). Если все указанные методы в данном состоянии для данных аргументов возвращают true, соответствующий тестовый метод выполняется в этом состоянии и с этими аргументами. Если хотя бы один метод из охранного условия возвращает false, соответствующий тестовый метод в этом состоянии и с этими аргументами не выполняется.

При выполнении данного теста с опцией -testclass mbtest.tests.guide.ListTest(4,4) выдается следующая трасса.

SUMMARY: Explorer: All is tested

SUMMARY: Explorer: Total number of failures = 0

SUMMARY: Explorer: Total number of states = 341

SUMMARY: Explorer: Total number of transitions = 3756

SUMMARY: Explorer: Total path length = 3756

SUMMARY: Explorer: Total time = 188

Количество состояний в построенном автомате равно 44+43+42+41+40 = (45-1)/(4-1) = 341 (каждое слагаемое — это количество списков определенной длины из четырех возможных элементов). Количество переходов — 44 \* (4\*2 + 0 + 0) + 43 \* (3\*2 + 3\*4 + 4) + 42 \* (2\*2 + 2\*4 + 4) + 41 \* (1\*2 + 1\*4 + 4) + 40 \* (0\*2 + 0\*4 + 4) = 256\*8 + 64\*22 + 16\*16 + 4\*10 + 1\*4 = 3756 (здесь первое слагаемое в скобках — количество переходов по remove() и get(), второе — по addInternal(), третье — по addLast()).

## Компоновка теста с использованием групп методов

## Настроечные методы

В ряде ситуаций необходимы специальные действия по инициализации теста или по освобождению ресурсов, захваченных в ходе его работы. Инструмент Summer поддерживает разнообразные настроечные методы, которые позволяют реализовать подобную функциональность.

Настроечные методы могут быть выполнены до или после начала выполнения тестового набора, до или после начала работы теста, до или после начала работы методов определенного тестового класса, до или после каждого метода в тесте, до или после методов определенной группы, а также при попадании в каждое состояние.

## Использование программных контрактов

В данном разделе описывается пример разработки теста с помощью Summer с использованием отдельных контрактов.

## Использование заглушек

## Компоновка сложных тестов

# Справочник по основным конструкциям

В данном разделе приведен краткий справочник по основным конструкциям (аннотациям языка Java и библиотечным классам), используемым при разработке тестов с помощью инструмента Summer.

## Параметры командной строки

Этот раздел содержит описание параметров командной строки, используемых при запуске класса mbtest.MBTester.

Для выдачи справочной информации служит опция -help или -?.

Поскольку полный набор опций командной строки может быть достаточно большим, произвольный их набор можно сохранять в файл. Для использования опций, описанных в файле с именем filename, в командной строке нужно задать опцию @filename.

### Описание компоновки выполняемых тестов

Задание других опций командной строки выглядит следующим образом.

COMMAND-LINE ::= (‘-testsuite’ ts-name TESTS-DESCRIPTION)+

COMMAND-LINE ::= TESTS-DESCRIPTION (‘-testsuite’ ts-name TESTS-DESCRIPTION)\*

COMMAND-LINE ::= TEST-OPTS TESTS-DESCRIPTION? (‘-testsuite’ ts-name TESTS-DESCRIPTION)\*

TESTS-DESCRIPTION ::= (‘-test’ t-name TEST-OPTS)+

TEST-OPTS ::= (‘-testclass’ TEST-OBJECTS)+ OTHER-TEST-OPTS

TEST-OBJECTS ::= Java-class-full-name (‘(‘ PARAMETERS ’)’)?( ‘;’ Java-class-full-name (‘(‘ PARAMETERS ’)’ )? )\*

PARAMETERS ::= param-value (‘,’param-value)\*

Смысл этих наборов опций следующий.

* Инструмент запускается для выполнения некоторого множества *тестовых наборов*. Каждый тестовый набор имеет имя и состоит из набора *тестов*, описание очередного тестовогонабора начинается опцией -testsuite, за которой следует имя этого набора и описание входящих в него тестов. Заканчивается оно там, где начинается описание другого тестового набора, или в конце командной строки.
* Описание теста начинается опцией -test, за которой следует имя теста, описание входящих в него тестовых объектов и опций данного теста. Заканчивается оно там, где начинается описание другого теста или тестового набора, или в конце командной строки.
* Первый тестовый набор может не иметь имени — тогда его описание не начинается с опции -testsuite, а просто состоит из описаний входящих тестов.
* Первый тест может не иметь имени — тогда его описание не начинается с опции -test, а просто состоит из описания входящих тестовых объектов и опций теста.
* Тестовые объекты описываются опцией -testclass, за которой следует список конструкторов тестовых объектов, перечисляемый через точку с запятой. Если для создания тестового объекта необходим вызов конструктора по умолчанию, достаточно указания полного имени класса, если нет — за именем класса должны в скобках следовать значения параметров одного из конструкторов через запятую (замечание: пробелов между скобками, запятыми и значениями параметров быть не должно!).  
  В описании теста может быть несколько описаний тестовых объектов — все эти объекты будут входить в данный тест.

В опциях командной строки обязательно указывать хотя бы один тестовый объект.

Опции одного теста описываются в следующих подразделах.

### Опции выбора тестовых методов

Каждый тест состоит из набора тестовых объектов, задающих его состояние.

Набор возможных действий теста задается тестовыми методами этих объектов, но не всегда все такие методы туда входят — опции выбора тестовых методов позволяют управлять этим.

Такие опции задаются перечислением имен групп методов через точку с запятой и включают следующие.

* Набор включаемых групп задается после опции -groups.
* Набор исключаемых грапп задается после опции -exclude.
* Набор оставляемых групп задается после опции -retain.

Если множества групп, заданные опциями -groups, -exclude, -retain обозначить через G, E, R, то результирующее множество методов, включаемых в тест, состоит из тех методов, которые входят в одну из групп множества G и в одну из групп множества R, но не входят ни в одну из групп множества E. Если опция -retain опущена, будут выбраны методы, входящие в одну из групп множества G и не входящие ни в одну из групп множества E.

Это правило отбора методов распространяется на тестовые методы, дескрипторы состояния и настроечные методы (т.е. все методы, имеющие специальные аннотации). Никак не затрагиваются вспомогательные методы, методы-источники данных, методы-охранные условия и пр. — участвуют ли они в тесте зависит только от того, будут ли они вызваны из отобранных методов (возможно, с использованием встроенных механизмов, как охранные условия).

### Конфигурация теста

Конфигурация теста включает, помимо набора входящих в него тестовых объектов, еще и некоторые вспомогательные объекты и набор связей между всеми этими объектами. Конфигурация теста в инструменте Summer задается XML-файлом с описанием этой конфигурации в формате, принятом в среде Spring. Framework. Правила описания конфигурации см. в следующем разделе.

Файл с описанием конфигурации теста задается при помощи указания -conffile filename.

Опция необязательна. Значения по умолчанию нет — у такого теста вся конфигурация описывается набором тестовых объектов.

### Уровень трассировки

Уровень трассирования в данном тесте задается при помощи указания -loglevel и желаемого уровня трассировки. Возможные уровни трассировки таковы (в порядке увеличения детальности, сообщения каждого уровня включаются во все следующие).

* fatal — только сообщения о сбоях в работе инстурмента;
* error — сообщения о всех обнаруженных ошибках и сбоях в работе инструмента;
* warning — сообщения о всех обнаруженных ошибках и предупреждения — сообщения о проблемных ситуациях, которые каким-либо образом разрешаются.
* trace — сообщения для подготовки отчетов (пока не реализовано).
* summary — итоговые информационные сообщения (итоги выполнения теста).
* info — все информационные сообщения (посещение состояний и выполнение переходов).
* debug — отладочная информация.

Опция необязательна. Значение по умолчанию — info.

### Используемый алгоритм обхода автомата

Алгоритм обхода, используемый в данном тесте, задается при помощи опции -explorer. Возможные ее значения таковы.

* none — обход не используется, состояния игнорируются.
* bfs — обход в ширину, рассчитан на детерминированные автоматы, требует линейный от числа состояний объем памяти (в простейших случаях ~1 MB на тысячу состояний, требования растут для сложно устроенных объектов-состояний, не учитывается память, необходимая для работы самой SUT), длина обхода близка к кратчайшей (при учете того фактора, что состояния и переходы становятся известны только в процессе построения обхода).
* oldbfs — обход в ширину без некоторых оптимизаций, требования аналогичные bfs, длина обхода немного длиннее (иногда в разы).
* local-greedy — локально жадный алгоритм, рассчитан на дельта-сильно связные недетерминированные автоматы (т.е., такие недетерминированные автоматы, в которых для любых двух состояний существует адаптивный путь, ведущий из первого во второе, адаптивность его означает, что он может по-разному продолжаться при попадании в различные состояния по одному и тому же воздействию из-за недетерминизма — это, по сути, не один путь, а множество путей, разветвляющихся в местах возможного недетерминизма, но начинающихся и заканчивающихся в одних и тех же состояниях), требует линейный от числа переходов объем памяти (в простейших случаях ~1 MB на тысячу переходов, требования растут для сложно устроенных объектов-состояний, не учитывается память, необходимая для работы самой SUT), длина обхода близка к кратчайшей (примерно такая же, как у bfs, иногда меньше, иногда больше).
* greedy — жадный алгоритм, рассчитан на детерминированные автоматы, требует квадратичный от числа состояний объем памяти (в 1.5 GB помещаются автоматы с ~5000 состояний в простейшем случае), длина обхода близка к кратчайшей (часто несколько меньше всех предыдущих), но время построения обхода растет быстрее, чем у предыдущих алгоритмов.
* dfs — обход в глубину, рассчитан на детерминированные автоматы, требует линейный от числа состояний объем памяти (часто значительно меньше, чем bfs), длина обхода обычно существенно больше кратчайшей, время построения обхода обычно растет быстрее, чем у предыдущих алгоритмов (оценка длины обхода и времени работы в худшем случае ровно такая же).
* olddfs — обход в глубину без некоторых оптимизаций, требования аналогичные dfs, длина обхода немного длиннее (иногда в разы).

Опция необязательна. Значение по умолчанию — bfs.

### Ограничения на тестовую последовательность

При построении тестовой последовательности (обходе автомата) действуют ограничения на сложность полученной последовательности. При превышении хотя бы одного из этих ограничений тест останавливается с выводом в трассу сообщения о причине остановки. Ограничения задаются следующими опциями.

* Ограничение на общую длину последовательности (число вызовов тестовых методов).  
  Задается опцией -maxpath. Значение по умолчанию — 20000.
* Ограничения на число различных переходов в автомате.  
  Задается опцией -maxtransitions. Значение по умолчанию — 5000.
* Ограничение на число различных состояний в автомате.  
  Задается опцией -maxstates. Значение по умолчанию — 500.

Отрицательное значение любой из этих опций означает, что соответствующее ограничение отсутствует.

Все эти опции необязательны.

### Ограничение на число обнаруженных ошибок

Ограничение на число обнаруживаемых тестом ошибок задается при помощи опции -maxfails.

Значение этой опции определяет максимальное число ошибок, при обнаружении которых тест будет пытаться восстановиться и продолжить работу. При превышении этого количества ошибок выполнение теста прекратится с выводом в трассу сообщения о причине остановки.

Отрицательное значение этой опции означает, что ограничения на количество ошибок нет.

Опция необязательна. Значение по умолчанию — 20.

### Политика восстановления после ошибок

Возможны две политики восстановления теста после ошибок.

* Игнорирование ошибок — после обнаружения ошибки работа теста продолжается без специфических действий.
* Восстановление начального состояния теста — после ошибки все тестовые объекты сбрасываются и строятся заново в исходной конфигурации, тест оказывается в начальном состоянии и пытается продолжить работу, используя ранее полученную информацию о состояниях и переходах и избегая выполнения того перехода, на котором была зафиксирована ошибка.

По умолчанию используется вторая политика.

При необходимости использовать первую политику нужно использовать опцию -ignorefails. Такая политика рекомендуется для тех случаев, где возможные ошибки никак не нарушают внутреннее состояние тестируемых объектов.

## Описание конфигурации теста

## Основные элементы теста

Данный раздел описывает аннотации, задающие основные элементы теста, или элементы структуры соответствующего ему расширенного автомата.

### Тестовый класс и тестовый метод

@Test

### Дескриптор состояния

@State

### Источник данных

@DataProvider

### Охранное условие

@Guard

## Настроечные элементы теста

@AfterMethod/@BeforeMethod

@AfterClass/@BeforeClass

@AfterTest/@BeforeTest

@AfterSuite/@BeforeSuite

@InState

## Библиотечные классы

Contract