Лабораторная работа №11

Цель работы:

освоить приемы тестирования кода на примере использования библиотеки JUnit

Требования к оформлению отчета

Отчет по лабораторной работе должен содержать следующие разделы (примеры оформления отчетов можно найти в папке с заданиями):

1) Изложение цели работы.
2) Задание по лабораторной работе с описанием своего варианта.
3) Спецификации ввода-вывода программы.
4) Текст программы (кратко).
5) Выводы по проделанной работе.

Методические указания

Что бы убедится, что программа работает корректно её работу необходимо протестировать. В самом простом случае тестирование программы заключается ее запуске и проверке результатов работы.

Такой способ применим для простых задач, но не для больших проектов. В них непротестированный код будет вызывать другой непротестированный код. И если где-либо будет ошибка, то найти её будет очень непросто. В этом случае нужно просмотреть код, изучить возможные пути исполнения, которые могли привести к ошибке, а потом эмпирически их выявлять.

Если программист хорошо знает свой код, то ошибка найдется скоро (при достаточном везении). Если код сторонний или слишком сложный, то это гарантирует долгие часы отладки.

Более продвинутые программисты будут пользоваться отладчиком для поиска ошибок, вставлять диагностические **System.out.prinltn(...)** в тело кода, или изучать журнал логов работы программы. Все эти подходы ориентированы на поиск уже существующих проблем, но не предотвращают появление ошибок в программе и даже не уменьшают их вероятность.

Чтобы уменьшить вероятность возникновения ошибки, а также ускорить ее поиск в программе, разработано много методологий программирования, способов организации кода, а также подходов к проведению тестирования. Одной из таких методологий является модульное тестирование. Основной принцип проведения модульного тестирования – разбиение программы на блоки, называемые модулями и проведение тестирования для каждого такого модуля по отдельности. Конечно, безупречная работа программы, составленной из протестированных модулей, не гарантируется, но можно утверждать, что она будет работать намного стабильнее.

В языке программирования Java в качестве модулей могут выступать:

- Классы
- Отдельные функции

Проблема уменьшения вероятности возникновения ошибок решается тем, что тесты пишутся до начала написания кода (методология разработки через тестирования – Test-Driven Development – TDD). Уменьшение же времени поиска ошибки достигается разумным покрытием кода достаточным количеством тестов, позволяющее быстро выявить проблемы на стадии разработки, а не при тестовом запуске в присутствии заказчика.

Задание 1 – Введение в JUnit

- Создаете новый класс и скопируйте код класса Sum;
- Создаете тестовый класс SumTest;
- Напишите тест к методу Sum.accum и проверьте его исполнение. Тест должен проверять работоспособность функции accum.
- Очевидно, что если передать слишком большие значения в Sum.accum, то случится переполнение. Модифицируйте функцию Sum.accum, чтобы она возвращала значение типа long и напишите новый тест, проверяющий корректность работы функции с переполнением. Первый тест должен работать корректно.

```
public class Sum {
   public static int accum(int... values) {
      int result = 0;
      for (int i = 0; i < values.length; i++) {
         result += values[i];
      }
      return result;
   }
}
```

Задание 2 – Тестирование функций

Подготовка к выполнению:

- Создайте новый проект в рабочей IDE;
- Создайте класс StringUtils, в котором будут находится реализуемые функции;
- Напишите тесты для реализуемых функций.

Написать тесты к методу, а затем реализовать сам метод по заданной спецификации. Варианты:

1) Реализуйте и протестируйте метод **String repeat(String pattern, int repeat)**, который строит строку из указанного паттерна, повторённого заданное количество раз.

   Спецификация метода:

```
repeat("e", 0) = ""
repeat("e", 3) = "eee"
repeat("ABC", 2) = "ABCABC"
repeat("e", -2) = IllegalArgumentException
repeat(null, 1) = NullPointerException
```

2) Разработайте метод String repeat(String str, String separator, int repeat), который строит строку из указанного паттерна, повторённого заданное количество раз, вставляя строку-разделитель при каждом повторении.

   Спецификация метода:

```
repeat("e", "|", 0) = ""
repeat("e", "|", 3) = "e|e|e"
repeat("ABC", ",", 2) = "ABC,ABC"
repeat("DBE", "", 2) = "DBEDBE"
repeat("DBE", ":", 1) = "DBE"
repeat("e", -2) = IllegalArgumentException
repeat("", ":", 3) = "::"
repeat(null, "a", 1) = NullPointerException
repeat("a", null, 2) = NullPointerException
```

3) Напишите метод **String keep(String str, String pattern)** который оставляет в первой строке все символы, которые присутствуют во второй.

   Спецификация метода:

```
keep(null, null) = NullPointerException
keep(null, *) = null
keep("", *) = ""
keep(*, null) = ""
keep(*, "") = ""
keep("hello", "hl") = "hll"
keep("hello", "le") = "ell"
```

4) Реализуйте функцию **String loose(String str, String remove)**, удаляющую из первой строки все символы, которые есть так же во второй.

   Спецификация метода:

```
loose(null, null) = NullPointerException
loose(null, *) = null
loose("", *) = ""
loose(*, null) = *
loose(*, "") = *
loose("hello", "hl") = "eo"
loose("hello", "le") = "ho"
```

5) Реализуйте и протестируйте метод **int indexOfDifference(String str1, String str2)** который сравнивает две строки и возвращает индекс той позиции, в которой они различаются. Например, **indexOfDifference("i am a machine", "i am a robot")** должно вернуть 7.

   Спецификация метода:

```
indexOfDifference(null, null) = NullPointerException
indexOfDifference("", "") = -1
indexOfDifference("", "abc") = 0
indexOfDifference("abc", "") = 0
indexOfDifference("abc", "abc") = -1
indexOfDifference("ab", "abxyz") = 2
indexOfDifference("abcde", "abxyz") = 2
indexOfDifference("abcde", "xyz") = 0
```

6) Напишите метод **String common(String str1, String str2)** сравнивающий две строки и возвращающий наибольшую общую часть.

   Спецификация метода:

```
common(null, null) = NullPointerException
common("", "") = ""
common("", "abc") = ""
common("abc", "") = ""
common("abc", "abc") = "abc"
common("ab", "abxyz") = "ab"
common("abcde", "abxyz") = "ab"
common("abcde", "xyz") = ""
common("deabc", "abcdeabcd") = "deabc"
common("dfabcegt", "rtoefabceiq") = "fabce"
```

7) Напишите метод **String substringBetween(String str, String open, String close)** выделяющий подстроку относительно открывающей и закрывающей строки.

   Спецификация метода:

```
substringBetween(null, null, null) = NullPointerException
substringBetween(null, *, *) = null
substringBetween(*, null, *) = null
substringBetween(*, *, null) = null
substringBetween("", "", "") = ""
substringBetween("", "", "]") = null
substringBetween("", "[", "]") = null
substringBetween("yabcz", "", "") = ""
substringBetween("yabcz", "y", "z") = "abc"
substringBetween("yabczyabcz", "y", "z") = "abc"
substringBetween("wx[b]yz", "[", "]") = "b"
```

8) Напишите метод **int levenshteinDistance(String s, String t)** рассчитывающий расстояние Ливенштейна для двух строк. Расстояние Ливенштейна между двумя строками – это то количество посимвольных трансформаций необходимое, что бы превратить одну строку в другую.

   Спецификация метода:

```
levenshteinDistance(null, null) = NullPointerException
levenshteinDistance(null, *) = -1
levenshteinDistance(*, null) = -1
levenshteinDistance("","") = 0
levenshteinDistance("","a") = 1
levenshteinDistance("aaapppp", "") = 7
levenshteinDistance("frog", "fog") = 1
levenshteinDistance("fly", "ant") = 3
levenshteinDistance("elephant", "hippo") = 7
levenshteinDistance("hippo", "elephant") = 7
levenshteinDistance("hippo", "zzzzzzzz") = 8
levenshteinDistance("hello", "hallo") = 1
```

9) Реализуйте метод **int hamingDistance(String str1, String str2)**, определяющий расстояние Хэминга для двух строк. Дистанция Хэминга – это число позиций, в которых соответствующие символы двух слов одинаковой длины различны.

   Спецификация метода:

```
hammingDistance(null, null) = NullPointerException
hammingDistance(null, *) = -1
hammingDistance(*, null) = -1
hammingDistance("abc", "abcd") = DifferentLengthException
hammingDistance("","") = 0
hammingDistance("father", "father") = 0
hammingDistance("pip", "pop") = 1
hammingDistance("abcd", "abab") = 2
hammingDistance("hello", "hallo") = 1
hammingDistance("abcd", "efgi") = 4
```

Задание 3 – Поиск ошибок, отладка и тестирование классов

1) Импорт проекта Импортируйте один из проектов по варианту:

   - **Stack** – проект содержит реализацию стека на основе связного списка: **Stack.java**.
   - **Queue** – содержит реализацию очереди на основе связного списка: **Queue.java**.

   Разберитесь как реализована ваша структура данных. Каждый проект содержит:

   - Клиент для работы со структурой данных и правильности ввода данных реализации (см. метод main()).
   - TODO-декларации, указывающие на нереализованные методы и функциональность.
   - FIXME-декларации, указывающую на необходимые исправления.
   - Ошибки компиляции (Синтаксические)
   - Баги в коде (!).
   - Метод check() для проверки целостности работы класса.

2) Поиск ошибок

   - Исправить синтаксические ошибки в коде.
   - Разобраться в том, как работает код, подумать о том, как он должен работать и найти допущенные баги.

3) Внутренняя корректность

   - Разобраться что такое утверждения (assertions) в коде и как они включаются в Java.
   - Заставить ваш класс работать вместе с включенным методом check.
   - Выполнить клиент (метод main() класса) передавая данные в структуру используя включенные проверки (assertions).

4) Реализация функциональности

   - Реализовать пропущенные функции в классе.
   - См. документацию перед методом относительно того, что он должен делать и какие исключения выбрасывать.
   - Добавить и реализовать функцию очистки состояния структуры данных.

5) Написание тестов

- Все функции вашего класса должны быть покрыты тестами.
- Использовать фикстуры для инициализации начального состояния объекта.
- Итого, должно быть несколько тестовых классов, в каждом из которых целевая структура данных создается в фикстуре в некотором инициализированном состоянии (пустая, заполненная и тд), а после очищается.
- Написать тестовый набор, запускающий все тесты.

Основные сведения о JUnit

Минимальный JUnit-тест имеет вид:

```
import org.junit.*; // Импорт всех основных классов и аннотаций JUnit
import static org.junit.Assert.*; // Импорт утверждений
// Тестовый сценарий
public class JUnitTestCase {
        // Отдельный тест
        @Test
        public void TestName() {
                // Проверка утверждений
        }
}
```

Терминология JUnit

Модель тестирования JUnit

- Тест в JUnit – это public и не static метод, помеченный аннотацией @Test (@org.junit.Test), содержащий одно или несколько утверждений и/или предложений о том, как должен вести себя тестируемый модуль.
- Утверждение (Assertion) – логические предположения, которые должны быть истинными в тесте. Если хотя бы одно из утверждений нарушается, тест считается проваленным.
- Тестовый класс, набор тестов (Test Case) – класс, содержащий один и более тестовых методов (помеченных аннотацией @Test). Тестовые классы используются для группировки тестов, тестирующих поведение одного модуля. Обычно один класс/метод имеет соответствующий ему тестовый класс.
- Тестовый набор (Test Suite) – группа тестов. Тестовый набор используется для группировки связанных тестов в единый набор, запускаемый одним сеансом. Например, нужно запустить все тесты, покрывающие функциональность некоторого пакета.

Основные утверждения в JUnit

- Утверждения истинности // Проверяет, что результат логического выражения вернул true assertTrue(<boolean expression>);
- Утверждения ложности // Проверяет, что результат логического выражения вернул false assertFalse(< expression>);

- Утверждение эквивалентности assertEquals(expected, actual);
- Утверждение неэквивалентности assertNotEquals(expected, actual);
- Утверждение эквивалентности массивов assertArrayEquals(expecteds[], actuals[]);

  Где:

  expected – ожидаемое значение тестируемого значения.

  actual – действительное значение тестируемого значения.

  Важное замечание: сравниваемыми аргументами в assertEquals могут быть и массивы. Их сравнение выполняется поэлементно.
- Null утверждения // Проверяет, что объект равен null assertNull(Object); // Проверяет, что объект не равен null assertNotNull(Object);
- Утверждения идентичности // Проверяет, что оба объекта идентичны друг другу assertSame(expected actual); // Проверяет, что оба объекта не идентичны друг другу assertNotSame(expected, actual);

Объявление тестов

Тестирование метода

Чтобы превратить метод в тест, достаточно добавить аннотацию @Test из пакета org.junit:

```
@Test
public void someTest() {
        ...
}
```

Тестирование исключений

Если надо проконтролировать появление ожидаемых исключений:

```
@Test(expected = SomeException.class)
public void someTest() {
        ...
}
```

Тестирование по времени выполнения

Если необходимо проконтролировать время выполнения теста (время указывается в миллисекундах):

```
@Test(timeout = 100)
public void someTest() {
        ...
}
```

Игнорирование тестов

Чтобы пропустить тест:

```
@Ignore
@Test
public void someTest() {
        ...
}
```

Возможные исходы теста

Тест завершается одним из следующих исходов (outcome):

- Успешно (Success) – все предположения выполнились, все утверждения пройдены успешно, неожиданных исключений не выброшено.
- Провал (Failure) – одно из утверждений внутри теста не выполнилось.
- Завершился с ошибкой (Error) – выполнение теста завершилось преждевременной по причине некоторой ошибки или исключения. Итоговый успех/провал теста не известен.
- Пропущен, проигнорирован (Ignore) – тест был проигнорирован, или не выполнилось одно из предположений, необходимых для выполнения теста.

Фикстуры

Сценарии тестов могут нуждаться в некоторых действиях, предшествующих тесту(наполнение БД) или выполняющихся после него(очистка БД). Для этого существуют так называемые фикстуры.

Выполнение кода до и после каждого теста

Чтобы определить код, который должен выполнятся перед каждым тестом из сценария:

```
@Before
public void setUpBeforTest() {
        ...
}
```

Код, выполняемый после исполнения каждого теста из сценария:

```
@After
public void setUpBeforTest() {
        ...
}
```

Группа тестов (Test Suite)

Тестовый набор (Test Suite) – группа тестов, запускаемая в одном сеансе. Например, нужно запустить все тесты, покрывающие функциональность некоторого пакета.

- Что бы создать тестовый набор в JUnit выберите некоторый класс, или создайте новый пустой класс.
- Пометьте класс аннотацией @RunWith(Suite.class), которая указывает библиотеке JUnit, что данный класс стоит обрабатывать как тестовый набор.
- Пометьте класс аннотацией @SuiteClasses(TestClass1.class, ...) в которой в фигурных скобках, через запятую перечислите все классы, входящие в данный тестовый набор.

Пусть, есть тестовые классы WcFileTest, WcDirTest, WcUnicodeTest, WcUnicodeTest. Тогда, объединение их в тестовый набор будет иметь вид:

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;
@RunWith(Suite.class) // Запустить класс как тестовый набор
@SuiteClasses({ // Список тестовых классов в наборе для запуска
        WcFileTest.class,
        WcDirTest.class,
        WcUnicodeTest.class,
        WcUnicodeTest.class
})
public class RunAll {
        // the class remains empty,
        // used only as a holder for the above annotations
}
```