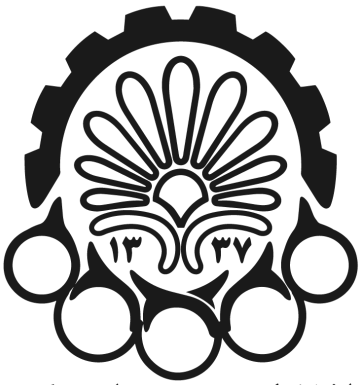


تمرین دوم درس تست نرم افزار

دانشکده مهندسی کامپیوتر، دانشگاه صنعتی امیرکبیر (پلی‌تکنیک تهران)

بردیا اردکانیان

۹۸۳۱۰۷۲

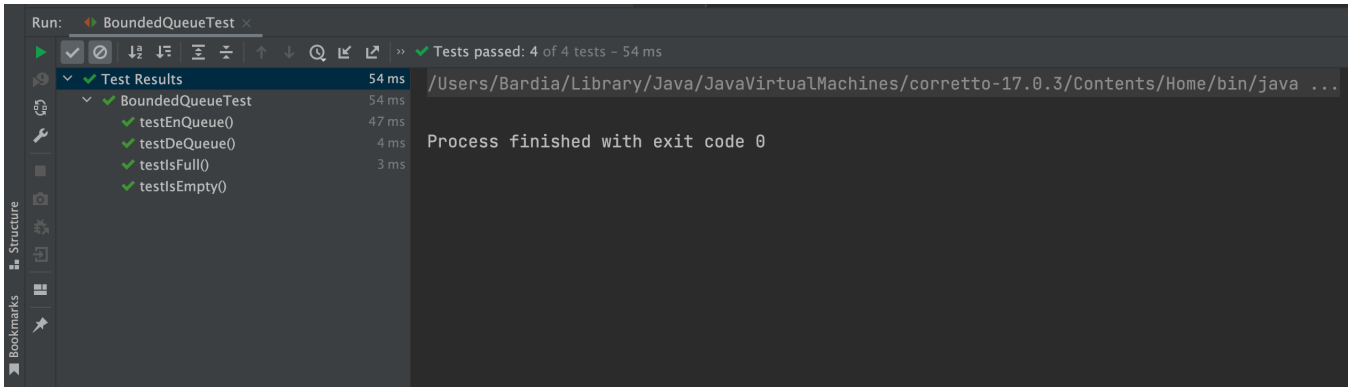
استاد درس: دکتر گوهری

بهار ۱۴۰۲

تمامی کدهای نوشته شده، ضمیمه فایل ارسالی شده‌اند و قابل اجرا می‌باشند.

Ch3-q3: Develop JUnit tests for the `BoundedQueue` class. A compilable version is available on the book website in the file `BoundedQueue.java`. Make sure your tests check every method, but we will not evaluate the quality of your test designs and do not expect you to satisfy any test criteria. Turn in a printout of your JUnit tests and either a printout or a screen shot showing the results of each test.

کد در فایل پاسخ به صورت zip شده قرار دارد. همچنین در صفحه بعدی کدها قابل مشاهده هستند.



```java
package com.company;

import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

public class BoundedQueueTest {

    private BoundedQueue queue;

    @BeforeEach
    void setUp() {
        queue = new BoundedQueue(3);
    }

    @Test
    void testEnQueue() {
        queue.enQueue("A");
        queue.enQueue("B");
        queue.enQueue("C");

        // Check that elements are added in the correct order
        Assertions.assertEquals("[A, B, C]", queue.toString());

        // Check that adding one more element throws an exception
        Assertions.assertThrows(IllegalStateException.class, () -> queue.enQueue("D"));
    }

    @Test
    void testDeQueue() {
        // Test that dequeueing an empty queue throws an exception
        Assertions.assertThrows(IllegalStateException.class, () -> queue.deQueue());

        // Test that dequeueing returns elements in the correct order
        queue.enQueue("A");
        queue.enQueue("B");
        queue.enQueue("C");
        Assertions.assertEquals("A", queue.deQueue().toString());
        Assertions.assertEquals("B", queue.deQueue().toString());
        Assertions.assertEquals("C", queue.deQueue().toString());

        // Test that dequeueing from an empty queue throws an exception
        Assertions.assertThrows(IllegalStateException.class, () -> queue.deQueue());
    }

    @Test
    void testIsEmpty() {
        // Test that a newly created queue is empty
        Assertions.assertTrue(queue.isEmpty());

        // Test that a non-empty queue is not empty
        queue.enQueue("A");
        Assertions.assertFalse(queue.isEmpty());

        // Test that an empty queue is still empty after dequeueing
        queue.deQueue();
        Assertions.assertTrue(queue.isEmpty());
    }

    @Test
    void testIsFull() {
        // Test that a newly created queue is not full
        Assertions.assertFalse(queue.isFull());

        // Test that a full queue is indeed full
        queue.enQueue("A");
        queue.enQueue("B");
        queue.enQueue("C");
        Assertions.assertTrue(queue.isFull());

        // Test that a non-full queue is still not full after dequeueing
        queue.deQueue();
        Assertions.assertFalse(queue.isFull());
    }
}
```

Ch3-q5: The following JUnit test method for the `sort()` method has a non- syntactic flaw. Find the flaw and describe it in terms of the RIPR model. Be as precise, specific, and concise as you can. For full credit, you must use the terminology introduced in the book.

In the test method, `names` is an instance of an object that stores strings and has methods `add()`, `sort()`, and `getFirst()`,

which do exactly what you would expect from their names. You can assume that the object `names` has been properly instantiated and the `add()` and `sort()` methods have already been tested and work correctly.

```
@Test
public void testSort()
{
  names.add ("Laura");
  names.add ("Han");
  names.add ("Alex");
  names.add ("Ashley");
  names.sort();
  assertTrue ("Sort method", names.getFirst().equals ("Alex"));
}
```

Assertion فقط بخش کوچکی از حالت نهایی (اولین عنصر در لیست) را بررسی می کند. بنابراین اگر یک ورودی باعث رخ دادن یک خطا شود و سپس به قسمت دیگری از حالت نهایی انتشار یابد، خطا آشکار نمی‌شود. تست باید به کل آرایه نگاه کند تا از وجود نداشتن خطا اطمینان حاصل کند.

Ch3-q6: Consider the following example class. `PrimeNumbers` has three methods. The first, `computePrimes()`, takes one integer input and computes that many prime numbers. `iterator()` returns an `Iterator` that will iterate through the primes, and `toString()` returns a string representation.

`computePrimes()` has a fault that causes it **not** to include prime numbers whose last digit is 9 (for example, it omits 19, 29, 59, 79, 89, 109, . ..). If possible, describe five tests. You can describe the tests as sequences of calls to the above methods, or briefly describe them in words. Note that the last two tests require the test oracle to be described.

(a) A test that does not reach the fault

```
@Test
void A() {
    PrimeNumbers primeNumbers = new PrimeNumbers();
    primeNumbers.computePrimes(5);

    Assertions.assertEquals("[2, 3, 5, 7, 11]", primeNumbers.toString());
}
```

(b) A test that reaches the fault, but does not infect

```
@Test
void B() {
    PrimeNumbers primeNumbers = new PrimeNumbers();
    primeNumbers.computePrimes(7);

    Assertions.assertEquals("[2, 3, 5, 7, 11, 13, 17]", primeNumbers.toString());
}
```

(c) A test that infects the state, but does not propagate

```java
@Test
void C() {
    PrimeNumbers primeNumbers = new PrimeNumbers();
    primeNumbers.computePrimes(7);

    Assertions.assertEquals("[2, 3, 5, 7, 11, 13, 17, 19]", primeNumbers.toString());
}
```

(d) A test that propagates, but does not reveal

```java
@Test
void D() {
    PrimeNumbers primeNumbers = new PrimeNumbers();
    int numPrimes = 10;

    primeNumbers.computePrimes(numPrimes);
    List<Integer> expectedPrimes = new ArrayList<>(primeNumbers.primes);

    primeNumbers.primes.removeIf(num -> num % 10 == 9);

    Iterator<Integer> iter = primeNumbers.iterator();
    List<Integer> actualPrimes = new ArrayList<>();
    while (iter.hasNext()) {
        actualPrimes.add(iter.next());
    }

    Assertions.assertEquals(expectedPrimes, actualPrimes);
}
```

(e) A test that reveals the fault

```java
@Test
void E() {
    PrimeNumbers primeNumbers = new PrimeNumbers();
    primeNumbers.computePrimes(7);

    Assertions.assertEquals("[2, 3, 5, 7, 11, 13, 17, 19]", primeNumbers.toString());
}
```

If a test cannot be created, explain why.

Ch3-q8: Develop a set of data-driven JUnit tests for the `Min` program. These tests should be for normal, not exceptional, returns. Make your `@Parameters` method produce both *String* and *Integer* values.

```java
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.CsvSource;
import static org.junit.jupiter.api.Assertions.assertEquals;

public class MinTest {

    @ParameterizedTest
    @CsvSource({
            "1, 2, 3, 1",
            "5, 10, 2, 2",
            "-5, -10, -2, -10",
            "0, 0, 0, 0",
            "1, 1, 1, 1"
    })
    void testMin(int a, int b, int c, int expected) {
        int[] arr = {a, b, c};

        int actual = Min.findMin(arr);
        assertEquals(expected, actual);
    }
}
```

Ch3-q9: When overriding the `equals()` method, programmers are also required to override the `hashCode()` method; otherwise clients cannot store instances of these objects in common `Collection` structures such as `HashSet`. For example, the `Point` class from Chapter 1 is defective in this regard.

(a) Demonstrate the problem with `Point` using a `HashSet`.

چون تابع hashcode را ننوشته‌ایم هردو را یک شی در نظر می‌گیرد.

```java
import java.util.HashSet;

public class Point {
    private int x;
    private int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    @Override
    public boolean equals(Object other) {
        if (!(other instanceof Point)) {
            return false;
        }
        Point otherPoint = (Point) other;
        return this.x == otherPoint.x && this.y == otherPoint.y;
    }

    public static void main(String[] args) {
        HashSet<Point> set = new HashSet<>();
        set.add(new Point(1, 2));
        set.add(new Point(3, 4));
        set.add(new Point(1, 2)); // This is a duplicate point
        System.out.println(set.size()); // Prints 3 instead of 2
    }
}
```

(b) Write down the mathematical relationship required between `equals()` and `hashCode()`.

رابطه مورد نیاز بین quals() و hashCode() به صورت زیر است:

- اگر دو شی مطابق با متد quals() برابر باشند، فراخوانی hashCode() بر روی هر یک از دو شیء باید یک عدد صحیح را ایجاد کند.
- اگر دو شی مطابق با متد quals() با هم برابر نباشند، فراخوانی hashCode() روی هر یک از دو شیء نیازی به تولید نتایج صحیح مجزا ندارد، اما توصیه می‌شود این کار را برای عملکرد بهتر مجموعه‌های مبتنی بر هش انجام دهید.

(c) Write a simple JUnit test to show that `Point` objects do not enjoy this property.

این تست دو شی Point با مقادیر x و y یکسان ایجاد می کند و آنها را به HashSet اضافه می کند. از آنجایی که متد equals() از Point مقادیر x و y را با هم مقایسه می کند، این دو شی باید برابر باشند. با این حال، از آنجایی که hashCode را override نمی‌کند، این دو شی کدهای هش متفاوتی خواهند داشت و HashSet آن‌ها را به‌عنوان اشیاء مجزا در نظر می‌گیرد و در نتیجه به جای ۱، اندازه مجموعه‌ای ۲ است.

```java
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;

import java.util.HashSet;
import java.util.Set;

public class PointTest {
    @Test
    public void testEqualObjectsHaveDifferentHashCodes() {
        Point p1 = new Point(1, 2);
        Point p2 = new Point(1, 2);

        Set<Point> pointSet = new HashSet<>();
        pointSet.add(p1);
        pointSet.add(p2);

        Assertions.assertEquals(2, pointSet.size());
    }
}
```

(d) Repair the `Point` class to fix the fault.

```java
public class Point {
    private final int x;
    private final int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    @Override
    public boolean equals(Object other) {
        if (!(other instanceof Point)) {
            return false;
        }
        Point p = (Point) other;
        return this.x == p.x && this.y == p.y;
    }

    @Override
    public int hashCode() {
        return Objects.hash(x, y);
    }
}
```

(e) Rewrite your JUnit *test* as an appropriate JUnit *theory*. Evaluate it with suitable `DataPoints`.

```java
import org.junit.experimental.theories.DataPoints;
import org.junit.experimental.theories.Theories;
import org.junit.experimental.theories.Theory;
import org.junit.runner.RunWith;

@RunWith(Theories.class)
public class PointTest {

    @DataPoints
    public static Point[] points = {
            new Point(0, 0),
            new Point(1, 1),
            new Point(2, 2),
            new Point(3, 3)
    };

    @Theory
    public void testEqualsAndHashCode(Point p1, Point p2) {
        Assume.assumeTrue(p1.equals(p2));
        assertTrue(p1.hashCode() == p2.hashCode());
    }
}
```

Ch4-q1: Chapter 3 contained the program `Calc.java`. It is available on the program listings page on the book website.
`Calc` currently implements one function: it adds two integers. Use test-driven design to add additional functionality to subtract two integers, multiply two integers, and divide two integers. First create a failing test for one of the new functionalities, modify the class until the test passes, then perform any refactoring needed. Repeat until all of the required functionality has been added to your new version of `Calc`, and all tests pass.

Remember that in TDD, the tests determine the requirements. This means you must encode decisions such as whether the division method returns an integer or a floating point number in automated tests **before** modifying the software.

Submit printouts of all tests, your final version of `Calc`, and a screenshot showing that all tests pass. Most importantly, include a narrative describing each TDD test created, the changes needed to make it pass, and any refactoring that was necessary.

ابتدا تست مربوط به تفریق را می‌نویسیم.

```
public void testSubtract() {
   int result = Calc.subtract(4, 2);
   assertEquals(2, result);
}
```

این تست بررسی می کند که روش تفریق به درستی تفاوت بین دو عدد صحیح را محاسبه می کند. وقتی این تست را اجرا می کنیم، با خطای کامپایل مواجه می شوید زیرا روش تفریق هنوز وجود ندارد. کد تفریق را اضافه می‌کنیم.

```
static public int subtract (int a, int b) {
   return a - b;
}
```

حالا وقتی تست را اجرا می کنید، باید قبول شود.

بریم سراغ ضرب. روش تست زیر را به کلاس CalcTest اضافه می‌کنیم:

```java
public void testMultiply() {
   int result = Calc.multiply(3, 4);
   assertEquals(12, result);
}
```

این تست بررسی می کند که آیا روش ضرب به درستی حاصل ضرب دو عدد صحیح را محاسبه می کند. وقتی این تست را اجرا می‌کنیم، با شکست مواجه می شود زیرا روش ضرب هنوز وجود ندارد. آن را به کلاس Calc اضافه می‌کنیم:

```java
static public int multiply (int a, int b) {
   return a * b;
}
```

حالا وقتی تست را اجرا می کنید، باید قبول شود. حالا یک تست برای تقسیم اضافه می‌کنیم. روش تست زیر را به کلاس CalcTest اضافه می‌کنیم:

```java
public void testDivide() {
   double result = Calc.divide(10, 3);
   assertEquals(3.3333, result, 0.0001);
}
```

این تست بررسی می کند که روش تقسیم به درستی ضریب دو عدد صحیح را محاسبه می کند. توجه داشته باشید که ما تلورانس ۰.۰۰۰۱ را برای نتیجه مورد انتظار تعیین کرده‌ایم، زیرا تقسیم ممکن است یک نتیجه ممیز شناور ایجاد کند که دقیقاً با مقدار مورد انتظار برابر نیست.

وقتی این تست را اجرا می کنید، به دلیل اینکه روش تقسیم هنوز وجود ندارد، با شکست مواجه می شود. آن را به کلاس Calc اضافه می‌کنیم:

```
static public double divide (int a, int b) {
   return (double) a / b;
}
```

کد نهایی

```
package com.company;

// Introduction to Software Testing
// Authors: Paul Ammann & Jeff Offutt
// Chapter 3; page ??
// See CalcTest.java, DataDrivenCalcTest.java for JUnit tests

public class Calc {
    public static int add(int a, int b) {
        return a + b;
    }

    public static int subtract(int a, int b) {
        return a - b;
    }

    public static int multiply(int a, int b) {
        return a * b;
    }

    public static double divide(int a, int b) {
        if (b == 0) {
            throw new IllegalArgumentException("Cannot divide by zero");
        }
        return (double) a / b;
    }
}
```

```java
package com.company;

import static org.junit.Assert.assertEquals;

import java.util.Arrays;
import java.util.List;
import java.util.function.IntBinaryOperator;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;
import org.junit.runners.Parameterized.Parameters;

@RunWith(Parameterized.class)
public class CalcTest {

    @Parameters
    public static List<Object[]> data() {
        return Arrays.asList(new Object[][] {
                { 0, 0, 0 },
                { 1, 2, 3 },
                { 2, 1, 3 },
                { -1, 2, 1 },
                { 2, -1, 1 },
                { -1, -2, -3 }
        });
    }

    private int a;
    private int b;
    private int expectedResult;

    public CalcTest(int a, int b, int expectedResult) {
        this.a = a;
        this.b = b;
        this.expectedResult = expectedResult;
    }

    @Test
    public void testAdd() {
        int result = Calc.add(a, b);
        assertEquals(expectedResult, result);
    }

    @Test
    public void testSubtract() {
        int result = Calc.subtract(a, b);
        assertEquals(expectedResult, result);
    }

    @Test
    public void testMultiply() {
        int result = Calc.multiply(a, b);
        assertEquals(expectedResult, result);
    }

    @Test
    public void testDivide() {
        double result = Calc.divide(a, b);
        assertEquals(expectedResult, result, 0.001);
    }

    static private int applyOperation (int a, int b, IntBinaryOperator op) {
        return op.applyAsInt(a, b);
    }
}
```