UNIVERSITÄT ZU LÜBECK
INSTITUTE FOR SOFTWARE ENGINEERING
AND PROGRAMMING LANGUAGES

isp

Development of Safety Critical Systems(Sichere Software) Winter term 18
Martin Leucker, Karam Kharraz, Felix Lange February 22, 2019

Exercise Sheet 2 (Solution)

Deadline: 15th/19th of November 2018

Task 2.1 Mutation-based Testing

The idea of mutation-based testing is to evaluate the quality of your test suite by inserting small changes in the code (mutations) and see if there is a test case that is able to detect this change as an error. We use the java mutation testing system pitest[1] which is working with JUnit.

Use the maven project in the Moodle course. It includes the code under test `triangleType`, which expects the three lengths of a triangle and returns the type of the triangle (1,2,3) or that the triangle is invalid (0). Also a JUnit test suite is provided.

```java
public Type triangleType(int a, int b, int c) {
  if (a<1 || b<1 || c<1){
    return Type.INVALID;
  }
  if ((a+b<=c) || (a+c<=b) || (b+c<a)) {
    return Type.INVALID;
  }
  if (a==b && b==c) {
    return Type.ISOSCELES;
  }
  if (a==b || a==c || b==c) {
    return Type.EQUILITERAL;
  }
  return Type.SCALENE;
}
```

Your task is to find the bug in code by using mutation-based testing:

a) Run the JUnit tests. Notice that there the coverage shown by EclEmma is not 100%.
b) Add some new test cases to reach full coverage.
c) Run pitest by executing `mvn org.pitest:pitest-maven:mutationCoverage` in path of the maven project.
d) Pitest should report, that at least one mutation has not been killed by the test cases. Try to figure out why this is the case and add new test cases that are able to detect the mutation.
e) Fix the bug in the code that should be noticed by the new test cases.
f) Run pitest again until there are no undetected mutations left.

Solution

a) Add some new test cases for coverage:

```java
assertEquals(Type.EQUILITERAL,codeUnderTest.triangleType(2,2,3));
assertEquals(Type.EQUILITERAL,codeUnderTest.triangleType(2,3,2));
assertEquals(Type.EQUILITERAL,codeUnderTest.triangleType(3,2,2));
```

[1] pitest.org

b) Fix Mutations:

```java
assertEquals(Type.INVALID,codeUnderTest.triangleType(2,1,1));
assertEquals(Type.INVALID,codeUnderTest.triangleType(1,2,1));
assertEquals(Type.INVALID,codeUnderTest.triangleType(1,1,2));
```

Task 2.2 Property-based Testing

Property-based Testing can be used to automatically generate big amounts of test cases and verify their output by checking their properties. In the Moodle course you can find a maven project which includes junit-quickcheck[2].

```java
List<Long> PrimeFactorization(Long i) {
  List<Long> primeFactors = new ArrayList<>();
  long divisor = 1;
  double squareRoot = Math.sqrt(i);
  while (i > 1) {
    divisor++;
    while (i % divisor == 0) {
      primeFactors.add(divisor);
      i /= divisor;
    }
    if (divisor > squareRoot) {
      divisor = i - 1;
    }
  }
  return primeFactors;
}
```

Your task is to verify the implementation of a function that returns the prime factors of the input number. One obvious property of this function is that all factors in the returned list are indeed prime numbers.

a) Create property-based test cases with junit-quickcheck that check if the returned list contains prime numbers only.

b) Think of two other properties that must be fulfilled for every input number.

Solution

```java
@Property
public void primeFactorsShouldBePrimeNumbers(
@InRange(min = "2", max = "999999999") Long a) {
  List<Long> factorsA = CodeUnderTest.PrimeFactorization(a);
  for(int i=0; i<factorsA.size(); i++) {
    assertTrue("All prime factors should be primes!",
    CodeUnderTest.isPrime(factorsA.get(i)));
  }
}

@Property
public void productOfPrimeFactorsShouldBeTheOriginalNumber(
@InRange(min = "1", max = "9999999999999") Long a) {
  List<Long> factorsA = CodeUnderTest.PrimeFactorization(a);
  Long expectedResult = (long) 1;
  for (int i = 0; i < factorsA.size(); i++) {
    expectedResult *= factorsA.get(i);
```

[2]https://github.com/pholser/junit-quickcheck

```java
  }
  assertEquals("The product of the prime factors ...!" ,expectedResult, a);
}

@Property
public void factorizationsAreUnique(@InRange(min = "2", max = "1000")
Long a, @InRange(min = "2", max = "1000") Long b) {
  assumeThat(a, not(equalTo(b)));
  List<Long> factorsA = CodeUnderTest.PrimeFactorization(a);
  List<Long> factorsB = CodeUnderTest.PrimeFactorization(b);

  assertThat(factorsA, not(equalTo(factorsB)));
}
```