Property Based Testing

- Properties
- Random Tests
- Type generators
- Shrinking
- Specifications as properties

Ciências ULisboa

slides: João Neto

The Pesticide Paradox

Every method you use to prevent or find bugs leaves a residue of subtler bugs against which those methods are ineffectual. -- Boris Beizer

- **Metaphor:** insects build resistance and pesticides no longer work
- When the test cycle is executed, failures are found and eventually corrected
- After more code is added, the same tests pick some extra failures that get corrected
  - But some new faults do not produce test failures
  - With time, if the test sets are the same, a population of resistant faults increase
- New tests need to be added
  - Some way to automate it?

Properties

- **Property: a true proposition given your programming context**
  - **Eg:** `product.weight() > 0`
  - **Eg:** `getGrade() >= 0 && getGrade() <= 20`
  - **Eg:** `list.length() == list.invert().length()`
  - **Eg:** `anOperation(itsInverse(x)) == x`
- **Properties represent domain knowledge, i.e., the business domain**
- The program should respect these properties, otherwise it is not correctly modelling the business domain
- Instead of checking actual values, we check properties

Property Based Testing

Assume we are testing an integer average function:

```
public static int average(int[] v) {
  if (v == null) throw new IllegalArgumentException();
  int r = 0;
  if (v.length > 0) {
    for (int i = 0; i < v.length; i++)
       r = r + v[i];
    r = r / v.length;
  }
  return r;
}
```

The typical Junit test would be:

```
@Test
public void testSimple() {
  int[] a = new int[] {1, 2, 3};
  int expected = 2;
  int actual = Average.average(a);

  assertEquals(expected, actual);
}
```

Property Based Testing

- But we know more about this function.
- For instance, we know that shuffling elements should not change the average result:

```
@Property
public void testShuffle(int[] xs) {
   assertEquals(Average.average(xs), Average.average(shuffle(xs)),
                    "average fails after shuffling");
}
```

- This method needs an array of ints to be injected
- What array?
- Who’s responsibe for the value injection?

Property Based Testing

- We will use **JUnit QuickCheck**, a Java library inspired on Haskell's QuickCheck
- This library will manage our properties and inject the needed values
- These values will be random, and the library will test each property multiple times!
- Repeating the same test many times with different values increases the probability of finding defects
- What is being tested by the following property?

```java
@Property(trials = 50)  // default is 100
public void testNegative(int[] xs) {
  int[] negatives = Arrays.stream(xs).map(i -> -i).toArray();

  assertEquals(-Average.average(xs), Average.average(negatives), "...");
}
```

Property Based Testing

- JUnit QuickCheck will generate random values given the type
- This means that very large values might be produced
- To test the property *"concatenating the array with itself does not change the average"*, we must be careful with overflows:

```
@Property
public void testDoublingVectorSize(
              @InRange(minInt = -1000, maxInt = 1000) int[] xs) {

  int[] xsTwice = IntStream
                  .concat(Arrays.stream(xs), Arrays.stream(xs))
                  .toArray();

  assertEquals(Average.average(xs), Average.average(xsTwice),
               "avg(xs++xs) != avg(xs)");
}
```

Generators

- JUnit QuickCheck allows the creation of new generators

```
public class SmallIntegerListGenerator extends Generator<List<Integer>> {

   public static final int MAX_SIZE = 100;
   public static final int MAX_INT = 1000;

   public SmallIntegerListGenerator(Class<List<Integer>> type) {
      super(type);
   }

   @Override
   public List<Integer> generate(SourceOfRandomness src, GenerationStatus
status) {
      int size = 1+src.nextInt(MAX_SIZE);
      LinkedList<Integer> list = new LinkedList<>();
      while(size-- > 0)
         list.add(src.nextInt(2*MAX_INT+1)-MAX_INT); // between [-1000,1000]
      return list;
   }
}
```

Generators

- Use the new generator via annotation `@From`

```
@Property
public void testDoublingVectorSize(
        @From(SmallIntegerListGenerator.class) List<Integer> list) {

  int[] xs = list.stream().mapToInt(i->i).toArray();

  int[] xsTwice = IntStream
                  .concat(Arrays.stream(xs), Arrays.stream(xs))
                  .toArray();

  assertEquals(Average.average(xs), Average.average(xsTwice),
               "avg(xs++xs) != avg(xs)");
}
```

- Exercise: create a generator for 2D points
- Exercise: create a string generator for strings with 40 chars having letters, digits and also `.-\;:_@[]^/|}{`

Generators

- `GenerationStatus` includes a map which can be used to pass information to another generator
- This eg produces a random list of decreasing values

```
public List<Integer> generate(SourceOfRandomness random, GenerationStatus status) {
  List<Integer> result = new ArrayList<>();

  int previous = status.valueOf(PREVIOUS_KEY).orElse(MAX_VALUE);
  int current = random.nextInt(previous);

  if (current > 0) {
    result.add(current);
    // use setValue(Key, Object) to pass values between generators
    status.setValue(PREVIOUS_KEY, current);
    // get one of the available generators using gen()
    Generator<List<Integer>> listGen = gen().make(DecreasingListGenerator.class);
    // status includes the actual max value defined above
    result.addAll(listGen.generate(random, status));
    status.setValue(PREVIOUS_KEY, null);
  }

  return result;
}
```

Ctor

- `Ctor` can be used to generate values from classes that have a single accessible constructor

```
public class Point2D {
  private int x, y;

  public Point2D(int x, int y) { this.x = x; this.y = y; }

  public int  getX()       { return x;   }
  public void setX(int x)  { this.x = x; }
  public int  getY()       { return y;   }
  public void setY(int y)  { this.y = y; }
  public String toString() { return "("+x+","+y+")"; }
}
```

```
@Property
public void testPointsGetSetXWithCtor(@From(Ctor.class) Point2D pt, int x) {
  pt.setX(x);
  assertTrue(pt.getX()==x);
}
```

Fields

- `Fields` can be used similarly for classes with a zero-arg constructor

```
public class Point2D {
  private int x, y;

  public int getX()         { return x;   }
  public void setX(int x)   { this.x = x; }
  public int getY()         { return y;   }
  public void setY(int y)   { this.y = y; }
  public String toString() { return "("+x+","+y+")"; }
}
```

```
@Property
public void testPointsGetSetYWithFields(@From(Fields.class) Point2D pt, int y) {
  pt.setY(y);
  assertTrue(pt.getY()==y);
}
```

ValuesOf

- `@ValuesOf` can be used to generate a Cartesian product of inputs (for booleans and enums)

```
enum Ternary { YES, NO, MAYBE }

@RunWith(JUnitQuickcheck.class)
public class CartesianProductTest {

  @Property(trials=6)
  public void testCartesianProduct(@ValuesOf boolean b, @ValuesOf Ternary t) {
    System.out.println(b+":"+t);
  }
}
```

```
false:YES
true:NO
false:MAYBE
true:YES
false:NO
true:MAYBE
```

A word about Randomness

- Randomness is an umbrella concept for many different phenomena
- There's no semantic in randomness without a proper discussion of the distribution of test data (aka, operational profile)
- By default the distribution is uniform over the type values
  - However this only makes sense for finite sets
  - Eg, for random strings or lists we must set a max size before generating values
- Random testing is more effective when the distribution of test data follows the distribution of actual data
  - Alas, in many cases the distribution of actual data is not known

Ok, some other words about Randomness

- Statistical independence between tests is very important
  - It allows for statistical predictions of significance in the observed results
  - Only random fluctuations (variance) can be averaged out, not systematics ones (bias)
  - This applies in testing: by dropping systematization of testing values, we minimize the change of bias to contaminate the test results
  - Variance can be dealt with just more tests
- But this might be pointless when the distribution of actual data is not known and it’s quite different from the distribution of test data

Features of Property Based Tests

- **Provides better parameter coverage**
- **Testing is incremental, building our trust; randomness provides new parameter values on each testing round**
- **Helps finding new edge cases we didn't think of (like weird strings, big negative numbers, …)**
- **A way to insert domain knowledge as a list of testable properties, i.e., a way to translate specification into the testing environment**
- **One property can replace many value examples**
- **D. Hamlet's, *Random Testing* (1994), quote:**

By taking 20% more points in a random test,
any advantage a partition test might have had is wiped out

Types of Properties

- **Properties of operations (eg, commutativity, associativity, neutral element)**
- **Inverse operations (eg, addition/subtraction, setters/getters, serialization/deserialization) – however two sequential errors can shadow a failure**
- **Invariants (collection size, same contents, balanced data structures)**
- **Idempotence** `op(x) == op(op(x))` **like sort, filter, unique, …**
- **For more general properties it is needed an Oracle, some process able to validate results**
  - **Usually it's not available, but…**
  - **We might have access to another algorithm that produces similar results (an older version, a less general version…)**
  - **Checking the results might be easy to code (eg, factoring a number can be easily checked by simple multiplication)**

Shrinking

- QuickCheck can do shrinking
- Shrinking means finding minimal counter-examples
- The idea is to start creating small random values and start increasing their complexity/size
- When a parameter value fails a test, the program tries to decrease it, depending on the type, to find another fail
  - Say the test fails at 998, then the program generates lots of numbers up to 998. If a test also fails at 685, it repeats the process until the shrinkage cannot find more counter-examples
- In QuickCheck to disable it use `@Property(shrink = false)`

Shrinking

- **Generators implement shrinking by overriding method doShrink**

```java
@Override
public List<Point2D> doShrink(SourceOfRandomness random,
                             Point2D larger) {

   if (ORIGIN.equals(larger))  // cannot shrink more
      return Collections.emptyList();

   List<Point2D> shrinks = new ArrayList<>();
   if (larger.getX()!=0) shrinks.add(new Point2D(0, larger.getY()/2));
   if (larger.getY()!=0) shrinks.add(new Point2D(larger.getX()/2, 0));
   if (larger.getX()!=0) shrinks.add(new Point2D(larger.getX()/4,
         larger.getY()));
   if (larger.getY()!=0) shrinks.add(new Point2D(larger.getX(),
         larger.getY()/4));
  return shrinks;
}
```

Specification as a Test Suite

- **A specification can be seen as a set of properties**
- **The testing suite can build tests focusing on what the program must comply**
- **It becomes a testable specification**
  - **eg, ADTs:**

```
L, L1: List[Element];    E, F: Element;

getFirst (addFirst (L, E)) = E;

removeFirst (addFirst (L, E)) = L;

isEmpty (L) iff size (L) = 0;

size (make ()) = 0;
size (addFirst (L, E)) = 1 + size (L);
```

Specification as a Test Suite

- **Each specification line will result in 1+ properties:**

```
@Property
public void testGetFirst(LinkedList<Integer> list, int e) {
    list.addFirst(e);

    assertEquals(list.getFirst(), e, "spec: getFirst(addFirst(L, E)) = E");
}

@Property
public void testRemoveFirst(LinkedList<Integer> list, int e) {
    LinkedList<Integer> newList = (LinkedList<Integer>) list.clone();
    newList.addFirst(e);
    newList.removeFirst();

    assertEquals(list, newList, "spec: removeFirst(addFirst(L, E)) = L");
}
```

- **Exercise: check** `sut.spec.List.spc` **and add more specification-based properties on file** `TestListSpec`

Exercise: Prime Factorization

- **Exercise: check class** `sut.PrimeFactors`
- **Create a test file to check the following properties:**
  - **The factors product should result in the original number**
  - **All factors must be prime numbers**
  - **For every factorization of an even number, the factors must include at least one number 2**

Exercise: Sale Generator

- **Check maven project**
- **In package** `sut.sale` **there's a set of classes implementing a basic Sale with Items**
- **Create a** `SaleGenerator` **to produce random sales. Output eg:**

```
[id: 0 date: 29/11/2016 items: (tangerine,140]
[id: 1 date: 14/05/2005 items: (pear,453) (orange,644) (banana,308)]
[id: 2 date: 16/07/2009 items: (pear,924) (apple,113)]
[id: 3 date: 03/10/2006 items: (apple,329) (watermelon,885) (apple,306)]
```

- Add the possibility to restrict the number of sale items using `@InRange` annotations

```
@Property(trials=15)
public void testSalesShow(@InRange(minInt = 1, maxInt = 50)
                          @From(SaleGenerator.class) Sale sale) {
   System.out.println(sale);
}
```

Exercise: Sale Generator

- **With the sale generator we can test properties.**
- **The following property asserts that sale items order is irrelevant to the sale total cost:**

```
@Property
public void testSalesItemOrder(@From(SaleGenerator.class) Sale sale) {
  int total = sale.getTotal();
  Collections.shuffle(sale.getItems());
  assertEquals(total, sale.getTotal(),
               "sale item order must be irrelevant to total cost");
}
```