10 Randomized Testing: QuickCheck

We now address the situation when we need to measure the performance of one program under a number of different circumstances. To get such measurements we need to run the program a large number of times on different data. Here the right approach is *randomized tests*.

Randomized tests run the test of some property of a program a large number of times on different data that is automatically generated. In order to be able to generate large number of random instances of a given type, we need a little bit of help.

The *QuickCheck for Java* library is one such tool.

Start a new project and download the files from the **TesterQuickCheck.zip**. Add to the project build paths both the *tester.jar* and *quickcheck-0.6.jar*.

A Simple Example

Open the file *Person.java*

We would like to check that when we define a class `Person` with two fields, `String first, String last,` and add methods `getFirst` and `getLast`, these methods will produce the correct values.

Here is the class `Person`:

```
class Person {
  private String first;
  private String last;

  public Person(String first, String last) {
    this.first = first;
    this.last = last;
  }

  public String getLast() { return last; }
  public String getFirst() { return first; }
}
```

Our test in the `Examples` class would be something like this:

```
public void testPerson(Tester t){
  Person johndoe = new Person("John", "Doe"):
  t.checkExpect(
      johndoe,
      new Person(johndoe.getFirst(),
                 johndoe.getLast()));
}
```

We have one test case. To make a new one, we need to come up with new first and last name and run a similar test again. The *quickcheck* library helps us here. It contains classes that generate random data for all primitive types. To generate a random `String`, we first define an instance of the primitive type generator for `String`s, then ask for the `next` instance of this type of data:

```
Generator<String> name = PrimitiveGenerators.strings();
```

We can now create a new instance of the class `Person` by providing to the constructor two new names as follows:

```
Person person = new Person(this.name.next(), this.name.next());
```

To generate an unlimited number of new instances of the class `Person` we include this code in the class `PersonGenerator` that implements the `net.java.quickcheck.Generator<Person>` interface. It has to define the method `next` that produces an instance of the type `Person`. Here is the complete class definition:

```
// the class that generates random instances of the Person class
class PersonGenerator
      implements net.java.quickcheck.Generator<Person>{
  Generator<String> name = PrimitiveGenerators.strings();

  @Override public Person next() {
    return new Person(name.next(), name.next());
  }
}
```

Our test will then be similar to the one above:

```
t.checkExpect(
    person,
    new Person(person.getFirst(),
               johndoe.getLast()));
```

However, the following code allows us to repeat this test 200 times. It includes a statement that prints the generated `String`s, so we can see what the data looks like.

```
public void testPerson(Tester t){
  System.out.println("Starting at 0");
  int i = 0;
  for(Person name : Iterables.toIterable(new PersonGenerator())){
    System.out.println(i + ":    " + name.getFirst() +
                           " ... " + name.getLast());
    t.checkExpect(
        name,
        new Person(name.getFirst(),
                   name.getLast()));
    i++;
  }
}
```

Look at the file **Person.java** and run the program. Look at the output – there are 200 test cases that have been run.

Generating one of several possible Values

Open the file *PersonPet.java*. We have extended the definition of the person to include the person's pet, but we allow only three kinds of pets: dogs, cats, and gerbils.

The class diagram for the class `PersonPet` is this:

```
 +--------------+
 | PersonPet    |
 +--------------+
 | String first |
 | String last  |
 | Pet pet      |--+
 +--------------+  |
                   |
         +---------+
         |
         v
    +-------------+
    | Pet         |
    +-------------+
    | String name |
    | String kind |
    +-------------+
```

However, we restrict the pet kinds, and so we do not want to generate random kinds of pets, only one of three possible choices. The `EnsuredValuesGenerator` will generate only one of the possible several values as follows:

```
  Collection<String> petkinds =
    Arrays.asList("cat", "dog", "gerbil");
  Generator<String> petkind =
    new EnsuredValuesGenerator<String>(this.petkinds);
```

Every time we invoke the method `petkind.next()` we will get one of the three possible `String`s: `"cat"`, `"dog"`, or `"gerbil"`.

Run the code for the file `PersonPet.java` and make sure you understand what is going on.

Generating integers within the given range

Let us now look at the classes in the file *Shapes.java*. We would like to generate new shapes, but would like them to represent objects within the bounds of our `Canvas`. The file *ShapeGenerators.java* shows us how this can be done:

```
class CartPtGenerator implements net.java.quickcheck.Generator<CartPt>{

  Generator<Integer> horizontal;
  Generator<Integer> vertical;

  CartPtGenerator(){
    this(400, 200);
  }

  CartPtGenerator(int w, int h){
    this.horizontal = new IntegerGenerator(0, w);
    this.vertical = new IntegerGenerator(0, h);
  }

  @Override public CartPt next() {
    return new CartPt(horizontal.next(), vertical.next());
  }
}
```

We see that the `IntegerGenerator` provided by the *quickcheck* library allows us to specify the bounds for the values of the integers it generates.

Read through the rest of the code, run it, and make sure you understand what is going on.

Exercise

We are now ready to see the real power of the randomized tests. Design the method `moveRandom` in the class `CartPt` that produces a new point moved by at most one pixel in any direction.

Now design a test that will check 200 times whether invoking the method `moveRandom` on an instance of `CartPt` produces a desired result.

Note 1: A lot of work goes into designing randomized tests. First we have to be able to generate data of the desired type. Next we need to understand and be able to define formally, as code, how the expected values relate to the object that invoked the method. We then need to have the testing support (such as the *tester* library, or `JUnit`) that will evaluate the tests and report the results. We then need to design the method that repeats the generation of the object to be involved in the test, the method invocation, and the testing of the outcome of the test invocation. And, of course, we need to run the tests.

Note 2: We have only recently started working with the *quickcheck* library. We discovered with a great surprise, that we are now able to easily generate such a large number of tests that the results do not fit within the *Console* window. We plan to add a new feature to the *tester* library that will save the test results to a file for reading later, or in some other way remedy this problem.