EN.601.422 / EN.601.622

Software Testing & Debugging

Plan for today

- Property-based Testing vs. Example-based Testing
- Jqwik Demo

Example-based Testing

```
import java.util.*;
import org.assertj.core.api.Assertions;
import org.junit.jupiter.api.Test;

class ListReverseTests {
    @Test
    void reverseList() {
        List<Integer> aList = Arrays.asList(1, 2, 3);
        Collections.reverse(aList);
        Assertions.assertThat(aList).containsExactly(3, 2, 1);
    }
}
```

But . . .

- How can I be confident that reverse also works with :
  - different values?
  - 5 elements?
  - 5000 elements?
  - an empty list?
  - elements of different types?
  - Etc.

Example-based Testing

- We have fought the fear with:
  - Input Space Partitioning
  - Boundary Value Analysis
  - Error Guessing
  - Structural Coverage (i.e., Whitebox testing)
  - Etc.

Property-based Testing

- We can approach the question of correctness from a different angle:

  Under what **preconditions** and constraints (e.g., the range of input parameters) should the functionality under test lead to which **postconditions** (results of a computation)? And which **invariants** should never be violated in the course?

Property

- What is a Property?

Combination of preconditions and qualities that are expected to be present in the code under test.

Reverse Example

- The *“reverse”* function:
  - For any given list of elements, applying reverse twice should result in the original list.
  - For any given list of elements, applying reverse must result in a list of the same exact size.
  - For any given list of elements, reverse of the list contains the same exact set of elements
  - Etc.

More Examples

- isPrime(int n):
  - *n* must be a positive value if it is prime
  - *n* must be an odd number if it is prime unless it is 2
  - If *n* is negative, then it is not prime
  - Etc.

More Examples

- areAnagrams(String s1, String s2):
  - If lengths of *s1* and *s2* are not equal, they cannot be anagrams
  - *s1* and *s2* must contain the same exact set of characters if they are anagrams
  - If *s1* and *s2* are equal, they are anagrams

Property-based Testing (PBT)

- The two main steps in PBT:
  1. **finding general and desired properties** for functions, components, and whole programs.
  2. **finding concrete inputs** to falsify those properties by the **randomized generation** of test data.

Property-based Testing

- QuickCheck originally for Haskell
- Many other tools have been developed since then to support various languages
  - Jqwik for Java: https://jqwik.net/docs/current/user-guide.html

Demo of Jqwik

@Forall

- Adding parameters and annotating them with **@ForAll** tells jqwik that you want the framework to generate concrete random input tests for you.

@ForAll Annotations

- Various annotations that can be used along with @Forall
  - @ForAll @StringLength(min = 1, max = 10) String string1
  - @ForAll @AlphaChars String string1
  - @ForAll @IntRange (int min = 0, int max = Integer.*MAX_VALUE*) int anInt
  - Etc.

Arbitrary<T>

- An interface that can be used to randomly choose a value from among a set of possibilities

Parameter Provisioning with @Provide

```
@Property
 boolean concatenatingStringWithInt(@ForAll("shortStrings") String aShortString,
                                    @ForAll("10 to 99") int aNumber ) {
   String concatenated = aShortString + aNumber;
   return concatenated.length() > 2 && concatenated.length() < 11;
 }
@Provide
Arbitrary<String> shortStrings() {
    return Arbitraries.strings().withCharRange('a', 'z')
    .ofMinLength(1).ofMaxLength(8); }
@Provide("10 to 99")
Arbitrary<Integer> numbers() {
    return Arbitraries.integers().between(10, 99);
}
```

Edge Cases

```
@Example
void printEdgeCases() {
    System.out.println(Arbitraries.integers().edgeCases());
    System.out.println(Arbitraries.strings().withCharRange('a', 'z').edgeCases());
    System.out.println(Arbitraries.floats().list().edgeCases());
}
```

```
EdgeCases[-2, -1, 0, 2, 1, -2147483648, 2147483647]
EdgeCases["a", "z", ""]
EdgeCases[[], [0.0], [1.0], [-1.0], [0.01], [-0.01], [-3.4028235E38], [3.4028235E38]]
```

Assumptions

```
@Property
boolean absOfNegativeNumbers(@ForAll @IntRange int a) {
    Assume.that(a > 0);
    return a == StringNIntUtil.abs(a);
}
```

Some Ideas for Finding Properties

- **Business rule:**
  - Example: For all customers with a yearly turnaround greater than X $ we give an additional discount of Y percent, if the invoice amount is larger than Z $.
- **Inverse functions:** applying the function first and the inverse function second should return the original
  - Example: adding and subtracting a fixed amount must not change a numeric value.
- **Idempotent functions:** multiple application of an idempotent function should not change results.
  - Example: Ordering a list multiple times, shouldn't change it after the first time.
- **Invariant functions:** some properties of code must be true at all times.
  - Example: sorting/mapping should never change the size of a collection
- **Fuzzing:** code should never explode, even if you feed it with lots of diverse and unforeseen input data.
  - No exceptions occur, at least no unexpected ones.
  - There are no 5xx return codes for HTTP requests; maybe you even require 2xx status all the time.
  - All return values are valid.
  - Runtime is under an acceptable threshold.

Summary

- PBT:
  - Pros:
    - gives us a new perspective to look at testing
    - relieves us from (some of) the burden of finding concrete inputs, corner cases and error guessing
  - Cons:
    - not exactly testing the actual output
    - brings a flair of indeterminism that random generation brings into the game.