UNIVERSITÀ
DEGLI STUDI DI BARI
ALDO MORO

SERLAB
Software Engineering Research

Integrazione e Test di Sistemi Software

Specification-based testing

Azzurra Ragone

Dipartimento di Informatica - Università degli Studi di Bari
Via Orabona, 4 - 70125 - Bari
Tel: +39.080.5443270 | Fax: +39.080.5442536
serlab.di.uniba.it

Specification-based testing: recap

7 steps to create the test suite:

1. Understanding the requirements (what the program must do, inputs, and outputs)
2. Explore what the program does for various inputs
3. Explore inputs, outputs and identify partitions
4. Identify boundary cases (aka corner cases)
5. Devise test cases
6. Automate test cases
7. Augment the test suite with creativity and experience

1) Understanding the requirements

Read the requirements carefully:

- what the program should do or NOT do?
- what are the inputs and the outputs?
- the types of variable involved (integers, strings, etc.)
- input domain (ex. $5<num<10$)
- some may be implicit (elicit them!)
- write it down!

2) Explore what the program does for various inputs

Especially important if you are not the one who writes the code

Build a mental model of the program

Play with the program: test it for different inputs

3) Explore inputs, outputs and identify partitions

Identify partitions:

- Look at each input individually:
  - identify the input **type** (int, string, etc.)
  - **range** of values (positive, negative, between two values, etc.)
  - **special** values (ex. Null)
- Look at dependencies among variables: how they interact with each other
- Explore the possible types of outputs

4) Identify boundary cases (aka corner cases)

Bugs love boundaries!

Identify boundaries of all the partitions.

5) Devise test cases

Testing **all combinations** of inputs could be expensive (and sometimes it is not possible).

Reduce the number of combinations.

Test **exceptional behavior only once** (not combine them).

6) Automate test cases

Write test in **JUnit**

Identify concrete **inputs** and know what you should expect as **output**

Writing tests is equivalent to writing code: tests should be easy to read and understand

It should be easy to understand what test failed and why

7) Augment the test suite with creativity and experience

Perform some **final checks**

Revisit all the tests you created to see if you miss some cases

Example: add two numbers

Implement `add()` method: receives two numbers, left and right (each represented as a list of digits), adds them, and returns the result as a list of digits.

Examples:

- [4,3] + [2,1] = [6,4]
- [2,5] + [1,8] = [4,3]

Requirements:

- Each element should be a number from [0–9]
- An `IllegalArgumentException` is thrown if this pre-condition does not hold

Example: add two numbers

```
public List<Integer> add(List<Integer> left, List<Integer> right) {
  if (left == null || right == null)
    return null;

  Collections.reverse(left);
  Collections.reverse(right);

  LinkedList<Integer> result = new LinkedList<>();

  int carry = 0;

  for (int i = 0; i < max(left.size(), right.size()); i++) {

    int leftDigit = left.size() > i ? left.get(i) : 0;
    int rightDigit = right.size() > i ? right.get(i) : 0;

    if (leftDigit < 0 || leftDigit > 9 ||
     rightDigit < 0 || rightDigit > 9)
      throw new IllegalArgumentException();

    int sum = leftDigit + rightDigit + carry;

    result.addFirst(sum % 10);

    carry = sum / 10;
  }

  return result;
}
```

Returns null if left or right is null

Reverses the numbers so the least significant digit is on the left

While there is a digit, keeps summing, taking carries into consideration

Throws an exception if the pre-condition does not hold

Sums the left digit with the right digit with the possible carry

The digit should be a number between 0 and 9. We calculate it by taking the rest of the division (the % operator) of the sum by 10.

If the sum is greater than 10, carries the rest of the division to the next digit

Let's look at the code for 5 mins and try to spot the bugs

Individual Inputs

`left` parameter:

1 - Empty
2 - Null
3 - Single digit
4 - Multiple digits
5 - Zeroes on the left

`right` parameter:

1 - Empty
2 - Null
3 - Single digit
4 - Multiple digits
5 - Zeroes on the left

Combinations of Inputs

`(left, right)` parameters:

1 - length (`left` list) > length (`right` list)
2 - length (`left` list) < length (`right` list)
3 - length (`left` list) = length (`right` list)

Are we missing something?

Even if not explicitly stated in the documentation: we should test cases with “**carry**”

[2,5] + [1,8] = [4,3]

It’s not enough to analyze parameters (and combinations of), you should also have a deep knowledge of the domain.

Test special case: carry

- Sum without a carry
- Sum with a carry: one carry at the beginning
- Sum with a carry: one carry in the middle
- Sum with a carry: many carries
- Sum with a carry: many carries, not in a row
- Sum with a carry: carry propagated to a new (most significant) digit (ex. 99 + 1 = 100) [boundary case]

Devise test cases

Pragmatically decide which partitions should be combined with others and which should not

Test <u>exceptional cases only once </u>and do not combine them (e.g. null, empty, single digit):

T1: `left` is null
T2: `left` is empty

T3: `right` is null
T4: `right` is empty

T5: single digit, no carry
T6: single digit, with carry

Devise test cases

Combinations of inputs

Multiple digits,
length (`left` list) = length (`right` list)

T7: no carry (22 + 33)
T8: carry in the least significant digit (29 + 23)
T9: carry in the middle (293 + 183)
T10: many carries (179 + 268)
T11: many carries, not in a row (19171 + 18161)
T12: carry propagated to a new (now most significant) digit (998 + 172)

Devise test cases

Combinations of inputs

Multiple digits,
length (`left` list) > length (`right` list)
OR
length (`left` list) < length (`right` list)

T13: no carry
T14: carry in the least significant digit
T15: carry in the middle
T16: many carries
T17: many carries, not in a row
T18: carry propagated to a new (now most significant) digit

Devise test cases

Special cases

Zeroes on the left (two cases are enough):
T19: no carry
T20: carry

Boundaries:
T21: carry to a new most significant digit, by one (ex. 99 +1).

ParameterizedTest

We are going to use the `ParameterizedTest` feature from JUnit:

- Write a test method `shouldReturnCorrectResult()` that works like a skeleton, with variables instead of hard-coded values

- Write a `testCases()` method that provide inputs to the `shouldReturnCorrectResult()`

- The link between the two methods is done through the `@MethodSource` annotation

ParameterizedTest

```
public class NumberUtilsTest {

  @ParameterizedTest
  @MethodSource("testCases")
  void shouldReturnCorrectResult(List<Integer> left,
   List<Integer> right, List<Integer> expected) {
    assertThat(new NumberUtils().add(left, right))
          .isEqualTo(expected);
  }
```

A parameterized test is a perfect fit for these kinds of tests!

Indicates the name of the method that will provide the inputs

Calls the method under test, using the parameterized values

ParameterizedTest

```
static Stream<Arguments> testCases() {

  return Stream.of(
    of(null, numbers(7,2), null), // T1
    of(numbers(), numbers(7,2), numbers(7,2)), // T2
    of(numbers(9,8), null, null), // T3
    of(numbers(9,8), numbers(), numbers(9,8 )), // T4

    of(numbers(1), numbers(2), numbers(3)), // T5
    of(numbers(9), numbers(2), numbers(1,1)), // T6
```

One argument per test case

Tests with nulls and empties

Tests with single digits

T1: `left` is null
T2: `left` is empty

T3: `right` is null
T4: `right` is empty

T5: single digit, no carry
T6: single digit, with carry

Software Engineering Research LABoratory

ParameterizedTest

```
static Stream<Arguments> testCases() {

  return Stream.of(
    of(null, numbers(7,2), null), // T1
    of(numbers(), numbers(7,2), numbers(7,2)), // T2
    of(numbers(9,8), null, null), // T3
    of(numbers(9,8), numbers(), numbers(9,8 )), // T4

    of(numbers(1), numbers(2), numbers(3)), // T5
    of(numbers(9), numbers(2), numbers(1,1)), // T6

    of(numbers(2,2), numbers(3,3), numbers(5,5)), // T7
    of(numbers(2,9), numbers(2,3), numbers(5,2)), // T8
    of(numbers(2,9,3), numbers(1,8,3), numbers(4,7,6)), // T9
    of(numbers(1,7,9), numbers(2,6,8), numbers(4,4,7)), // T10
    of(numbers(1,9,1,7,1), numbers(1,8,1,6,1),
      numbers(3,7,3,3,2)), // T11
    of(numbers(9,9,8), numbers(1,7,2), numbers(1,1,7,0)), // T12
```

One argument per test case

Tests with nulls and empties

Tests with single digits

Tests with multiple digits

length (`left` list) = length (`right` list)

T7: no carry (22 + 33)

T8: carry in the least significant digit (29 + 23)

T9: carry in the middle (293 + 183)

T10: many carries (179 + 268)

T11: many carries, not in a row (19171 + 18161)

T12: carry propagated to a new (now most significant) digit (998 + 172)

ParameterizedTest

Multiple digits,
length (`left` list) > length (`right` list)
OR
length (`left` list) < length (`right` list)

T13: no carry
T14: carry in the least significant digit
T15: carry in the middle

Tests with multiple digits, different length, with and without carry (from both sides)

```
of(numbers(2,2), numbers(3), numbers(2,5)), // T13.1
of(numbers(3), numbers(2,2), numbers(2,5)), // T13.2
of(numbers(2,2), numbers(9), numbers(3,1)), // T14.1
of(numbers(9), numbers(2,2), numbers(3,1)), // T14.2
of(numbers(1,7,3), numbers(9,2), numbers(2,6,5)), // T15.1
of(numbers(9,2), numbers(1,7,3), numbers(2,6,5)), // T15.2
```

ParameterizedTest

Multiple digits,
length (`left` list) > length (`right` list)
OR
length (`left` list) < length (`right` list)

T16: many carries
T17: many carries, not in a row
T18: carry propagated to a new (now most significant) digit

Tests with multiple digits, different length, with and without carry (from both sides)

```
of(numbers(3,1,7,9), numbers(2,6,8), numbers(3,4,4,7)), // T16.1
of(numbers(2,6,8), numbers(3,1,7,9), numbers(3,4,4,7)), // T16.2
of(numbers(1,9,1,7,1), numbers(2,1,8,1,6,1),
  numbers(2,3,7,3,3,2)), // T17.1
of(numbers(2,1,8,1,6,1), numbers(1,9,1,7,1),
  numbers(2,3,7,3,3,2)), // T17.2
of(numbers(9,9,8), numbers(9,1,7,2), numbers(1,0,1,7,0)), // T18.1
of(numbers(9,1,7,2), numbers(9,9,8), numbers(1,0,1,7,0)), // T18.2
```

Software Engineering Research LABoratory

ParameterizedTest

Zeroes on the left:
T19: no carry
T20: carry

Boundaries:
T21: carry to a new most significant digit, by one (ex. 99 +1).

Tests with zeroes on the left

```
  of(numbers(0,0,0,1,2), numbers(0,2,3), numbers(3,5)), // T19
  of(numbers(0,0,0,1,2), numbers(0,2,9), numbers(4,1)), // T20

  of(numbers(9,9), numbers(1), numbers(1,0,0)) // T21
 );
}
```

The boundary test

Test Results

Tests that fail:

- All the tests that deal with a carry that become a new leftmost digit fail
(ex. 9 + 2 = 11 returns 1 or
998 + 172 = 1170 returns 170)

- Test with zeroes on the left (result expected [3,5], result returned [0,0,0,3,5]) fail

```
> shouldThrowExceptionWhenDigitsAreOutOfRange(List, List)
v shouldReturnCorrectResult(List, List, List)
    [1] null, [7, 2], null
    [2] [], [7, 2], [7, 2]
    [3] [9, 8], null, null
    [4] [9, 8], [], [9, 8]
    [5] [1], [2], [3]
    [6] [9], [2], [1, 1]
    [7] [2, 2], [3, 3], [5, 5]
    [8] [2, 9], [2, 3], [5, 2]
    [9] [2, 9, 3], [1, 8, 3], [4, 7, 6]
    [10] [1, 7, 9], [2, 6, 8], [4, 4, 7]
    [11] [1, 9, 1, 7, 1], [1, 8, 1, 6, 1], [3, 7, 3, 3, 2]
    [12] [9, 9, 8], [1, 7, 2], [1, 1, 7, 0]
    [13] [2, 2], [3], [2, 5]
    [14] [3], [2, 2], [2, 5]
    [15] [2, 2], [9], [3, 1]
    [16] [9], [2, 2], [3, 1]
    [17] [1, 7, 3], [9, 2], [2, 6, 5]
    [18] [9, 2], [1, 7, 3], [2, 6, 5]
    [19] [3, 1, 7, 9], [2, 6, 8], [3, 4, 4, 7]
    [20] [2, 6, 8], [3, 1, 7, 9], [3, 4, 4, 7]
    [21] [1, 9, 1, 7, 1], [2, 1, 8, 1, 6, 1], [2, 3, 7, 3, 3, 2]
    [22] [2, 1, 8, 1, 6, 1], [1, 9, 1, 7, 1], [2, 3, 7, 3, 3, 2]
    [23] [9, 9, 8], [9, 1, 7, 2], [1, 0, 1, 7, 0]
    [24] [9, 1, 7, 2], [9, 9, 8], [1, 0, 1, 7, 0]
    [25] [0, 0, 0, 1, 2], [0, 2, 3], [3, 5]
    [26] [0, 0, 0, 1, 2], [0, 2, 9], [4, 1]
    [27] [9, 9], [1], [1, 0, 0]
```

Bugs fix

- Simple fix for the carry bug: add the carry at the end

```java
int carry = 0;
for (int i = 0; i < Math.max(left.size(), right.size()); i++) {

    int leftDigit = left.size() > i ? left.get(i) : 0;
    int rightDigit = right.size() > i ? right.get(i) : 0;

    if (leftDigit < 0 || leftDigit > 9 || rightDigit < 0 || rightDigit > 9)
        throw new IllegalArgumentException();

    int sum = leftDigit + rightDigit + carry;

    result.addFirst( e: sum % 10);
    carry = sum / 10;
}

// if there's some leftover carry, add it to the final number
if (carry > 0)
    result.addFirst(carry);
```

Bugs fix

- Simple fix for zeroes on the left bug: remove the zeroes on the left before returning the result

- Ex. left = [0,0,0,1,2]
  right = [0,2,3]
  result expected [3,5]
  result returned [0,0,0,3,5].

Bu

-

be

```
for (int i = 0; i < Math.max(left.size(), right.size()); i++) {

    int leftDigit = left.size() > i ? left.get(i) : 0;
    int rightDigit = right.size() > i ? right.get(i) : 0;

    if (leftDigit < 0 || leftDigit > 9 || rightDigit < 0 || rightDigit > 9)
        throw new IllegalArgumentException();

    int sum = leftDigit + rightDigit + carry;

    result.addFirst( e: sum % 10);
    carry = sum / 10;
}

// if there's some leftover carry, add it to the final number
if (carry > 0)
    result.addFirst(carry);

// remove leading zeroes from the result
while (result.size() > 1 && result.get(0) == 0)
    result.remove( index: 0);

return result;
```

Ex.:
[0,0,0,3,5]

Are we missing something?

Requirements:
- Each element should be a number from [0–9]
- An `IllegalArgumentException` is thrown if this pre-condition does not hold

```
@ParameterizedTest
@MethodSource("digitsOutOfRange")
void shouldThrowExceptionWhenDigitsAreOutOfRange(List<Integer> left,
  ➥ List<Integer> right) {
  assertThatThrownBy(() -> new NumberUtils().add(left, right))
      .isInstanceOf(IllegalArgumentException.class);

}

static Stream<Arguments> digitsOutOfRange() {
  return Stream.of(
      of(numbers(1,-1,1), numbers(1)),
      of(numbers(1), numbers(1,-1,1)),
      of(numbers(1,10,1), numbers(1)),
      of(numbers(1), numbers(1,11,1))
  );
}
```

A parameterized test also fits well here.

Asserts that an exception happens

Passes invalid arguments

Lessons learned

The bug found in this example where due not to incorrect code, but to *lack of code*.

These are the type of bugs discovered by specification-based testing.

Here enforcing “code-coverage” would have been useless.

Take away lessons

- *Requirements* are super-useful to generate tests
- It's important to know the *domain space* and how the variables interact
- Go for the simplest input (ex. pick a small `int` value or a `short` string)
- Follow the *7 steps approach*
- Remember that bugs love *boundaries*
- If the number of test cases is too large, you should decide what is worth testing and what is not (what would be the cost of a failure?)
- Use *parameterized tests* when tests have the same skeleton

Boundary cases: Q1

Condition: value > 100
What are the on and off points?

Time: 1 minutes

Slide courtesy of Mauricio Aniche

Boundary cases: Q1

Condition: value > 100
What are the on and off points?

On point = 100 (always the value in the condition)
The on-point makes the condition false ((100 > 100) == false), therefore the off-point should make the condition true.
Off point = 101

Slide courtesy of Mauricio Aniche

Boundary cases: Q2

Condition: value >= 101

What are the on and off points?

Time: 1 minutes

Slide courtesy of Mauricio Aniche

Boundary cases: Q2

Condition: value >= 101

What are the on and off points?

On point = 101 (always the value in the condition)

The on-point makes the condition true ((101 >= 101) == true), therefore the off-point should make the condition false.

Off point = 100

Slide courtesy of Mauricio Aniche

Boundary cases: Q3

Condition: value == 100
What are the on and off points?

Time: 1 minutes

Slide courtesy of Mauricio Aniche

Boundary cases: Q3

Condition: value == 100
What are the on and off points?

On point = 100 (always the value in the condition)
The on-point makes the condition true.
There are two off-points!
Off point = 101 and 99

Slide courtesy of Mauricio Aniche

Boundary cases: Q4

Condition: value > n + 1
What are the on and off points?

Time: 2 minutes

Slide courtesy of Mauricio Aniche

Boundary cases: Q4

Condition: value > n + 1
What are the on and off points?

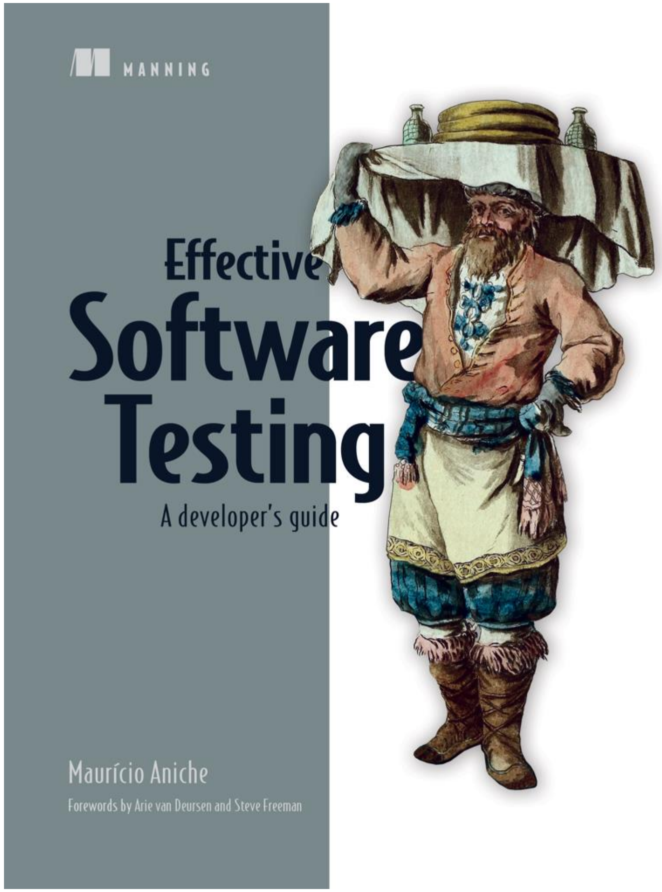
On point = n + 1 (always the value in the condition)
The on-point makes the condition false (n + 1 > n + 1 == false).
The off-point should make the condition true
Off point = (n + 1) + 1
If n is an input parameter, then, you would pick any "n" and then "n+1".

Slide courtesy of Mauricio Aniche

Reference book:

Effective Software Testing. A developer’s guide. Mauricio Aniche. Ed. Manning. (**Chapter 2**)

Use the "au35ani" discount code for a 35% off the price.



Software Engineering Research LABoratory

References:

- AssertJ - fluent assertions java library: https://assertj.github.io/doc/
- Assertj core javadoc: https://www.javadoc.io/doc/org.assertj/assertj-core/latest/index.html
- Assertj core javadoc: Assertions: https://www.javadoc.io/doc/org.assertj/assertj-core/latest/org/assertj/core/api/Assertions.html
- Introduction to AssertJ: https://www.baeldung.com/introduction-to-assertj

Azzurra Ragone

Department of Computer Science- Floor VI – Room 616
Email: azzurra.ragone@uniba.it